AD-A118 813   RHODE ISLAND UNIV KINGSTON DEPT OF COMPUTER SCIENCE --ETC   F/G 9/2
ALGORITHMIC COMPLEXITY. VOLUME I.(U)
JUN 82   E A LAMAGNA, L J BASS, L A ANDERSON   F30602-79-C-0124
UNCLASSIFIED   81-161-VOL-1   RADC-TR-82-152-VOL-1   NL

1 of 2

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-82-152, Vol I (of two)	2. GOVT ACCESSION NO. AD-A118813	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ALGORITHMIC COMPLEXITY		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report June 1979 - August 1981
		6. PERFORMING ORG. REPORT NUMBER 81-161
7. AUTHOR(s) Edmund A. Lamagna Ralph E. Bunker Leonard J. Bass Philip J. Janus Lyle A. Anderson		8. CONTRACT OR GRANT NUMBER(s) F30602-79-C-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Phode Island Dept of Computer Science and Statistics Kingston RI 02881		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101F LD9202C1
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COEE) Griffiss AFB NY 13441		12. REPORT DATE June 1982
		13. NUMBER OF PAGES 132
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Joseph P. Cavano (COEE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Analysis of Algorithms
Computational Complexity
Software Quality Metrics
Efficiency

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The objective of this study was to conduct applied research directed toward understanding the relationship between the complexity or efficiency of algorithms and the overall quality of computer software. The final report is presented in a two volume series consisting of a total of eight parts. This volume, containing Parts 1 and 2, comprises a general introduction to the entire series and a research plan.

Part 1 begins with a description of the goals of the overall contract

Cont'd

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Cont'd

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

effort. This is followed by a discussion of previous RADC work on software quality metrics, emphasizing measures concerned with the time and storage efficiency of programs. Next, an overview of the field of algorithm analysis and computational complexity is given. A final section contains an introduction to the particular research investigations pursued in the other portions of this study.

Part 2 presents a research plan for advancing the state-of-the-art in the area of algorithm performance. The executional behavior of a problem is a complex function of the efficiency of the underlying algorithm, the programming language used to implement it, the efficiency of the code produced by the compiler, the speed and architecture of the hardware, and features of the operating system. While recommending continuing work of an applied nature in algorithm analysis and computational complexity, new and unresolved issues concerning the relationships between programming languages, computer architecture, and the performance of algorithms on computer systems are also identified. Problem application areas considered include computational algebra; sorting, searching, and data base systems; pattern matching in strings; combinatorial optimization problems; and computational geometry.

An Appendix contains an outline of the major topics and issues addressed in the area of algorithm analysis and computational complexity, together with an annotated select bibliography.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

PREFACE

This is the first of two volumes constituting the final technical report for a study entitled "Algorithmic Complexity". The work was performed in support of the Information Sciences Division, Rome Air Development Center, under U.S. Air Force Systems Command contract F30602-79-C-0124. The duration of the project was from June 1979 through August 1981.

The research described herein was performed by members of the Department of Computer Science and Experimental Statistics at the University of Rhode Island. Dr. Edmund A. Lamagna served as Principal Investigator for this effort. Dr. Leonard J. Bass was Co-Principal Investigator. Three graduate assistants -- Messrs. Lyle A. Anderson, Ralph E. Bunker, and Philip J. Janus -- also worked on the project. Technical guidance was provided by Mr. Joseph P. Cavano, RADC Project Engineer.

The study consists of eight parts, whose titles are:

1. Measures of Algorithmic Efficiency: An Overview (Lamagna)
2. The Performance of Algorithms: A Research Plan (Lamagna, Bass, and Anderson)
3. Fast Computer Algebra (Lamagna)
4. Systematic Analysis of Algorithms (Anderson)
5. Adaptive Methods for Unknown Distributions in Distributive Partitioning Sorting (Janus)
6. Expected Behavior of Approximation Algorithms for the Euclidean Traveling Salesman Problem (Lamagna with E. J. Carney and P. V. Kamat)
7. Data Base Access Methods (Bass)
8. An Experimental Evaluation of the Frame Memory Model of a Data Base Structure (Bunker and Bass)

Volume I contains Parts 1 and 2, comprising a general introduction to the entire series and a research plan. Volume II contains the remaining six parts, describing the results of several technical investigations which were conducted.

Abstract

This document is an introduction to, and overview of, an Algorithmic Complexity contract effort performed at the University of Rhode Island (URI) for Rome Air Development Center (RADC). The objective of the study was to conduct applied research for understanding the relationship between the efficiency of algorithms and the overall quality of computer software.

The paper begins with a description of the specific missions of the overall contract effort. This is followed by a discussion of previous RADC work on software quality metrics, including a critical evaluation of the measures relating to the time and storage efficiency of programs. Next, a general overview of the field of algorithm analysis and computational complexity is given. Several shortcomings in the nature of current algorithmic complexity research are identified. These perspectives provide the rationale for the particular research directions pursued in this study. The final section of the paper is a brief introduction to each of these investigations, which are detailed in the other seven parts of this series.

The Appendix contains an outline of the major topics and issues addressed in the area of algorithm analysis anc computational complexity, together with an annotated select bibliography referencing materials which cover and survey most of the work performed to date.

TABLE OF CONTENTS

LIST OF TABLES

SECTION 1

INTRODUCTION

The objective of this study was to conduct applied research for the development of techniques for understanding the relationship between the complexity of algorithms and the overall quality of computer software. Measurement of software quality is made in terms of factors in which cost and time considerations are used to provide a baseline for evaluating the factors. These factors include correctness, reliability, efficiency, integrity, usability, maintainability, testability, flexibility, portability, reusability, and interoperability.

Work in the area of analysis of algorithms and computational complexity is directed principally at the factor of software efficiency, while insuring that correctness is not sacrificed. The goal of this work is to predict the behavioral characteristics, particularly time and storage utilization, of the software which will result by using certain algorithms. These predictions provide a design tool whereby alternative algorithms can be compared, both against each other and against lower bounds on the resources known to be needed in order to solve the problem. As a result, the algorithm which best meets the requirements for the application at hand can be selected for implementation. A useful by-product of such analyses is that algorithms can often be improved or new, more suitable algorithms developed. All of this leads to software of both greater efficiency and higher quality.

The thrust of this effort centered on investigating and developing techniques and measurements which can be used to evaluate the complexity of algorithms and the software which results from their use. Specifically, the tasks to be performed as delineated in the contract's Statement of Work, were as follows:

1) Perform a survey of algorithms for common problems, ranking the amounts of computational resources used with the best existing lower bounds to date. The survey should present a comprehensive picture of the state-of-the-art in the area of algorithm complexity.
2) Develop a research plan, both on a long and short term basis, for advancing the state-of-the-art and solving the problems of algorithm complexity. The plan should address the amount and kind of research that is needed in this area.
3) Study the inherent computational complexity of common algorithms and show how this complexity gets generated into a computer system design and eventually a computer program.
4) Develop techniques for analyzing algorithms. These techniques should be based on the control structures used and should provide a better understanding of computer program behavior. Study the process of automating this type of analysis.
5) Determine metrics for software quality factors that are related to the complexity of the algorithms used. The

metrics should fit into RADC's ongoing program on software quality.

6) Determine quantitative design tradeoffs between important system parameters, such as storage and time requirements, for certain algorithms.

7) Develop techniques for quantitatively studying the influence of the total computing environment (i.e.. both hardware and software) on the design and implementation of algorithms.

The remainder of this paper consists of three major sections. The first of these discusses previous RADC work on software quality metrics, including a critical evaluation of the measures which relate to efficiency. The next section provides an overview to the field of algorithm analysis and computational complexity. Several shortcomings in the nature and direction of current algorithmic complexity research are identified.

These perspectives provide the rationale behind the particular research directions we chose to pursue in this study. The final section of the paper gives a brief introduction to and perspective on these investigations, which are detailed in the other parts of this series.

SECTION 2

RADC SOFTWARE QUALITY METRICS

This study is an outgrowth of prior RADC work on software quality measurement. The Air Force is constantly striving to improve the quality of its software systems. High quality software is necessary to satisfy the stringent performance, reliability, and error-free requirements of software for Command and Control, as well as other application areas. To help accomplish these objectives, precise definitions of software quality are needed. Based on this framework, metrics quantifying software quality for objective analysis can be derived. Establishment of such measures should have a beneficial impact on the implementation and evaluation of a software product at each stage of its development.

In an initial RADC study, McCall, Richards, and Walters [1a] identified and defined the following eleven factors affecting software quality:

. correctness - extent to which a program satisfies its specification and fulfills the user's mission objectives

. reliability - extent to which a program can be expecte to perform its intended function with required precision

. efficiency - the amount of computing resources and code required by a program to perform a function

. integrity - extent to which access to software or data by unauthorized persons can be controlled

- usability - effort required to learn, operate, prepare input, and interpret output of a program
- maintainability - effort required to locate and fix an error in an operational program
- testability - effort required to test a program to insure it performs its intended function
- flexibility - effort required to modify an operational program
- portability - effort required to transfer a program from one hardware configuration and/or software system environment to another
- reusability - extent to which a program can be used in other applications; related to the packaging and scope of the functions that programs perform
- interoperability - effort required to couple one system with another

The above software quality factors are user-oriented by nature. The study went on to identify specific criteria, or attributes of the software or software production process, in terms of which these factors can be judged. Examples include error tolerance, consistency, accuracy, and simplicity for reliability; and generality, modularity, software system independence, machine independence, and self-descriptiveness for reusability. The two criteria associated with efficiency, the factor of primary interest here, were execution and storage efficiency.

Based upon such criteria, McCall, Richards, and Walters proposed a number of metrics, both objective and subjective, for measuring software quality. The units of the metrics were generally chosen as the ratio of actual occurrences to the number of possible occurrences of some attribute. (E.g., for keeping loop invariant computations outside of loops, the measure

$$1 - \frac{\text{\# loop dependent statement in loop}}{\text{total \# loop statements}}$$

was used). When this was not feasible, 0-1 measures based on the absence or presence of a characteristic were used. (E.g., a performance optimizing compiler was used.) The measures relating to efficiency are presented in Table 5 and discussed later in this section.

The original RADC study also included a data collection and validation effort for the metrics based on software development data from actual Air Force systems [1b]. A preliminary handbook for software acquisition managers on using the metrics was also prepared [1c]. A second, follow-on study for RADC was conducted by McCall and Matsumoto [2a]. This effort went on to refine and enhance the software quality measurement process proposed in the initial study. The work also included an analysis of metric applications, and a further validation of some of the metrics using actual software development data. The study also produced a Software Quality Measurement Manual [2b] containing procedures and guidelines for assisting software system developers in setting quality goals, applying the proposed metrics, and making quality assessments.

In the remainder of this section, we discuss and critique the previous RADC work on software quality metrics as it relates to efficiency. We begin by discussing the relative importance of efficiency, and the relationship between efficiency and the ten other quality factors. Next, we examine some of the efficiency measures which were proposed, commenting on their applicability and suitability. Finally, we give a critical assessment of these measures.

Just how important is efficiency, relative to the other software quality factors, to various Air Force applications? As part of both quality metrics efforts, several people familiar with Air Force missions were asked to identify the importance of each of the factors to the software produced. The results of the first survey [1a] are presented in Table 1 by specific mission, and those of the second effort [2a] are shown in Table 2 grouped by application area. In the first survey, efficiency received an overall rating of "high" importance, with only correctness and reliability receiving a higher overall rating ("high" to "very high"). In the second survey, efficiency received a "high" overall score in the Comm^nd and Control area, "high" to "medium" for the Indications and Warning and Simulation areas, and "medium" to "low" for Support Software.

What can be concluded from these surveys is that the relative importance of the quality factors varies according to the application environment (with the exception of correctness and reliability which achieved consistently "high" to "very high" ratings). Efficiency is an important requirement in many

Table 1

IMPORTANCE OF SOFTWARE QUALITY FACTORS TO SPECIFIC AIR FORCE APPLICATIONS

		FACTORS										
		PRODUCT OPERATION					PRODUCT REVISION			PRODUCT TRANSITION		
EXAMPLE SYSTEMS	EXPLANATION	CORRECTNESS	RELIABILITY	EFFICIENCY	INTEGRITY	USABILITY	MAINTAINABILITY	FLEXIBILITY	TESTABILITY	PORTABILITY	REUSABILITY	INTEROPERABILITY
RADC H6000/NSW	Development/Test Bed/R&D Lab type System	L-M	M	L	L-M	H	L-M	H	M	H	H	L
Graduate Pilot Flight Simulator	Pilot Training	H	H	H	L	H-V	M	M-H	M	L	L	L
Personnel System at AF Base	Personnel status, assignments, promotions, etc.	H	M	L-M	H	H	M-H	L	M	L-M	M	L-M
MACIMS - Military Airlift Command Information Management System	Allocation of Aircraft	H	H	H	M-H	H	H	M	M	M	M	H
SATIN IV-SAC Total Information Network	World Wide Communication System	H	H-V	H-V	V	H-V	H	M	H	L	L	M
E-4 Advanced Airborne Command Post	Survivable C^2 Airborne Operation	H-V	V	H-V	H-V	H-V	H	H	H	L	L	H
Over the Horizon Radar	NORAD surveillance and warning system. Detect targets at all altitudes.	H-V	V	H	H	H	H	M-H	H	L	L	M
TIPI - Tactical Info Processing & Interpretation System	Intelligence data processing for the tactical commander. Air transportable mobile shelters.	H	V	H	V	H	H	H	H	M	M	H
TRACALS - Traffic Control and Landing Systems	Air traffic control including Terminal Navigation Aids, Landing Systems, Air Traffic Control Simulators	V	V	H	M-H	V	H	M	H	L-M	L	M
MINUTEMAN	ICBM Offensive Weapon System	V	V	H-V	H	L-M	H	M	V	L	L	L
DMSP - Defense Meteorological Satellite Program	Weather, Mapping Applications	H	H	M-H	M	M	H	M	H	L	L	L
DSP - Defense Support Program	Defense Oriented Satellites	V	V	H	H-V	H	H-V	H	V	L	L-M	M
STS - Space Transportation System (SHUTTLE)	Manned Spacecraft - High Visibility Program	V	V	H	H	V	H-V	H	V	L	L	L-M
FB-111 Medium Bomber Airborne Avionics	Includes navigation and weapons control, inertial navigation, terrain following radar, etc.	V	V	H-V	H	H	H	H	H-V	L	L	L
AVERAGE RATING OF FACTORS		H-V	H-V	H	H	H	H	M	H	L	L	L-M

LEGEND

Level of importance of quality factors.

V - Very High

H - High

M - Medium

L - Low

From: McCall, Richards, and Walters, "Factors in Software Quality", RADC-TR-77-369, Vol. I

Table 2

RELATIVE IMPORTANCE OF SOFTWARE QUALITY FACTORS

AVERAGE AND STANDARD DEVIATION BY APPLICATION AREA

CORRECTNESS		4	0
RELIABILITY		3.75	0.5
TESTABILITY		3.75	0.5
FLEXIBILITY	COMMAND AND CONTROL	3.25	0.5
EFFICIENCY		3	0.82
MAINTAINABILITY	4 SYSTEMS	3	0.82
USABILITY		2.5	0.58
INTEGRITY		2	1.15
REUSABILITY		1.5	1
INTEROPERABILITY		1.5	1
PORTABILITY		1.25	0.5
CORRECTNESS		3.88	0.45
RELIABILITY		3.75	0.53
MAINTAINABILITY	INDICATIONS	3.13	0.85
INTEROPERABILITY	AND	3.04	1.08
USABILITY		3	0.88
TESTABILITY	WARNING	3	0.88
FLEXIBILITY		2.92	0.88
INTEGRITY	24 SYSTEMS	2.79	1.02
EFFICIENCY		2.75	0.85
PORTABILITY		1.92	0.97
REUSABILITY		1.71	0.75
CORRECTNESS		4	0
RELIABILITY		4	0
USABILITY		3.6	0.55
MAINTAINABILITY		3.2	0.45
TESTABILITY		2.8	0.45
FLEXIBILITY	SIMULATION	2.8	1.1
EFFICIENCY		2.4	0.55
INTEGRITY	5 SYSTEMS	2	1.41
INTEROPERABILITY		1.8	1.3
PORTABILITY		1.4	0.55
REUSABILITY		1	0
USABILITY		3.83	0.41
RELIABILITY		3.5	0.55
MAINTAINABILITY		3.5	0.55
PORTABILITY		3.5	0.84
CORRECTNESS		3.33	0.52
REUSABILITY	SUPPORT SOFTWARE	3.33	0.82
FLEXIBILITY		3.17	0.41
TESTABILITY	6 SYSTEMS	2.5	0.55
INTEROPERABILITY		2.5	1.22
EFFICIENCY		1.67	0.82
INTEGRITY		1.33	0.82

From: McCall and Matsumoto, "Software Quality Metrics Enhancements", RADC-TR-80-109, Vol. I software systems, particularly those dealing with Command and Control, Communications, real-time applications, and in systems with limited available resources (i.e., slow processors, small memories). In such systems, efficiency can become an overriding factor which, if not present, might possibly render the system useless. For example, failure to meet timing requirements in certain military contexts can result in the loss of life, materiel, or inability to make a critical decision on time. On the other hand, efficiency tends to be less important in data processing environments, such as Management Information Systems, where one does not usually need to respond immediately to the information and reports produced.

What is the relationship between efficiency and the other software quality factors? McCall, Richards, and Walters discuss this question in [1a], and Tables 3 and 4 summarize their conclusions. One would expect most of the quality factors to have a high positive correlation. For example, one would obviously expect portable systems to be highly reusable, and that the properties of software which make it reusable will also aid in making it portable. Good documentation and structured coding practices contribute to the correctness, testability, usability, and maintainability of a system, so one would also expect a direct relationship between these factors.

Unfortunately, efficiency appears to have a negative impact on most of the other quality factors (all except correctness and reliability), as seen in Table 3. This is most disconcerting because it is the only quality factor which exhibits this

Table 3

RELATIONSHIPS BETWEEN SOFTWARE QUALITY FACTORS

FACTORS	CORRECTNESS	RELIABILITY	EFFICIENCY	INTEGRITY	USABILITY	MAINTAINABILITY	TESTABILITY	FLEXIBILITY	PORTABILITY	REUSABILITY	INTEROPERABILITY
CORRECTNESS											
RELIABILITY	○										
EFFICIENCY											
INTEGRITY			●								
USABILITY	○	○	●	○							
MAINTAINABILITY	○	○	●		○						
TESTABILITY	○	○	●		○	○					
FLEXIBILITY	○	○	●	●	○	○	○				
PORTABILITY			●			○	○				
REUSABILITY		●	●	●		○	○	○	○		
INTEROPERABILITY			●	●					○		

LEGEND

If a high degree of quality is present for factor, what degree of quality is expected for the other:

○ = High ● = Low

Blank = No relationship or application dependent

From: McCall, Richards, and Walters, "Factors in Software Quality", RADC-TR-77-369, Vol. I

Table 4

SOFTWARE QUALITY TRADEOFFS INVOLVING EFFICIENCY

INTEGRITY VS EFFICIENCY	The additional code and processing required to control the access of the software or data usually lengthens run time and require additional storage.
USABILITY VS EFFICIENCY	The additional code and processing required to ease an operator's tasks or provide more usable output usually lenghten run time and require additional storage.
MAINTAINABILITY VS EFFICIENCY	Optimized code, incorporating intricate coding techniques and direct code, always provides problems to the maintainer. Using modularity, instrumentation,and well commented high level code to increase the maintainability of a system usually increases the overhead resulting in less efficient operation.
TESTABILITY VS EFFICIENCY	The above discussion applies to testing.
PORTABILITY VS EFFICIENCY	The use of direct code or optimized system software or utilities decreases the portability of the system.
FLEXIBILITY VS EFFICIENCY	The generality required for a flexible system increases overhead and decreases the efficiency of the system.
REUSABILITY VS EFFICIENCY	The above discussion applies to reusability.
INTEROPERABILITY VS EFFICIENCY	Again the added overhead for conversion from standard data representations, and the use of interface routines decreases the operating efficiency of the system.

From: McCall, Richards, and Walters, "Factors in Software Quality", RADC - TR-77-369, Vol. I property to any significant degree. It makes it seem as though virtually all of the other quality factors must be sacrificed if one desires efficiency.

The reasons for this conclusion arise from McCall, Richards, and Walters taking a "low level" view of software efficiency. That is, they are concerned with the efficiency of the object code rather than with efficiency at the level of algorithm design. They assume that the algorithm to be used is somehow given, and that efficiency involves fine tuning the algorithm to increase its speed or decrease its storage requirements at the implementation phase. In this study, we are concerned with increasing efficiency by making the most suitable choice of algorithm at the design phase, before any coding begins.

The rationale behind McCall, Richards, and Walters conclusions is summarized in Table 4. The basic paradigm behind their reasoning goes as follows. The traditional activities of the software development cycle include first designing a solution to a problem, then coding it, and finally testing the resultant implementation for correctness and to ascertain performance data. Sometimes efficiency is an important design factor, or an initial version of the software may fail to meet the timing and storage utilization requirements in the system specification. In such cases, the programs may be coded, or recoded, using intricate coding techniques or using assembly language instead of a high-level language. Such techniques have inherent difficulties associated with them in terms of the increased effort needed to write, debug, maintain, and transport the resulting software.

In the paradigm described above, little thought is given to reexamining the underlying algorithms used, and perhaps researching or developing alternative approaches. Programmers generally use only the standard algorithms with which they are familiar or, when faced with a novel application, use a direct "brute force" approach. Instead, algorithmic complexity focuses attention on the overall design of a software system. By suitably formulating and modeling the problem to be solved, alternative algorithms for its solution can be studied and the best available one selected. The choice of algorithm can then aid in the selection of an appropriate programming language and implementation techniques. The advantages of this approach are that the performance of the selected algorithm can be estimated before the coding and testing effort actually begins, and design tradeoffs can be considered more quantitatively. Efficiency need not have a negative impact on the other software quality control factors if such an approach is taken!

The efficiency measures proposed by McCall, Richards, and Walters [1a] are presented in Table 5. Two execution efficiency metrics, one dealing with iterative processing and the other with data usage, and a storage efficiency metric were developed. Each of the three metrics is an average of 5 to 11 elemental scores. The metrics may be applied to either a single module or an entire software system. The measurements are made during the design, coding, and debugging phases of the software cycle,

Table 5a

EXECUTION EFFICIENCY METRICS

FACTOR(S): EFFICIENCY

CRITERION/ SUBCRITERION	METRIC	REQMTS YES/NO 1 OR 0	REQMTS VALUE	DESIGN YES/NO 1 OR 0	DESIGN VALUE	IMPLEMENTATION YES/NO 1 OR 0	IMPLEMENTATION VALUE
EXECUTION EFFICIENCY/ REQUIREMENTS	EE. 1 PERFORMANCE REQUIREMENTS ALLOCATED TO DESIGN			☐			
	SYSTEM METRIC VALUE = Same as line above				☐		
ITERATIVE PROCESSING	EE. 2 ITERATIVE PROCESSING EFFICIENCY MEASURE: (by module)						
	(1) Non-loop dependent computations kept out of loop. $\left(1 - \frac{\text{\# nonloop dependent statements in loop}}{\text{total \# loop statements}}\right)$				☐		☐
	(2) Performance optimizing compiler/assembly language used.					☐	
	(3) Compound expressions defined once. $\left(1 - \frac{\text{\# compound expression defined more than once}}{\text{\# compound expressions}}\right)$						☐
	(4) Number of overlays. $\left(\frac{1}{\text{\# of overlays}}\right)$				☐		☐
	(5) Free of bit/byte packing/unpacking in loops.			☐		☐	
	(6) Free of nonfunctional executable code. $\left(1 - \frac{\text{\# nonfunctional executable code}}{\text{total executable statements}}\right)$						☐
	(7) Decision statements efficiently coded. $\left(1 - \frac{\text{\# inefficient decision statements}}{\text{Total \# decision statements}}\right)$						☐

From: McCall, Richards, and Walters, "Factors in Software Quality", RADC-TR-77-369, Vol. I

Table 5b

EXECUTION EFFICIENCY METRICS (CONTINUED)

FACTOR(S): EFFICIENCY

CRITERION/ SUBCRITERION	METRIC	REQMTS		DESIGN		IMPLEMENTATION	
		YES/NO 1 OR Ø	VALUE	YES/NO 1 OR Ø	VALUE	YES/NO 1 OR Ø	VALUE
	(8) Module linkages. $\left(1 - \frac{\text{module linkage time}}{\text{execution time}}\right)$						☐
	(9) OS linkages. $\left(1 - \frac{\text{OS linkage time}}{\text{execution time}}\right)$						☐
	MODULE METRIC VALUE = $\frac{\text{total score from applicable elements}}{\text{total \# applicable elements}}$				☐		☐
	SYSTEM METRIC VALUE = $\frac{\text{sum of iterative processing measures for each module}}{\text{total \# modules}}$				☐		☐
DATA USAGE	EE. 3 DATA USAGE EFFICIENCY MEASURE: (by module)						
	(1) Data grouped for efficient processing.			☐		☐	
	(2) Variables initialized when declared. $\left(\frac{\text{\# initialized when declared}}{\text{total \# variables}}\right)$						☐
	(3) No mix-mode expressions. $\left(1 - \frac{\text{\# mix mode expressions}}{\text{\# executable statements}}\right)$						☐
	(4) Common choice of units/type. (1/# occurrences of uncommon unit operations)				☐		☐
	(5) Data indexed or referenced for efficient processing.			☐		☐	
	MODULE METRIC VALUE = $\frac{\text{total score from applicable elements}}{\text{\# applicable elements}}$				☐		☐
	SYSTEM METRIC VALUE = $\frac{\text{sum of data usage measures for each element}}{\text{total \# modules}}$				☐		☐

Table 5c

STORAGE EFFICIENCY METRICS

FACTOR(S): EFFICIENCY

CRITERION/ SUBCRITERION	METRIC	REQMTS		DESIGN		IMPLEMENTATION	
		YES/NO 1 OR Ø	VALUE	YES/NO 1 OR Ø	VALUE	YES/NO 1 OR Ø	VALUE
STORAGE EFFICIENCY	SE. 1 STORAGE EFFICIENCY MEASURE: (by module)						
	(1) Storage requirements allocated to design.			☐			
	(2) Virtual storage facilities used.			☐		☐	
	(3) Common data defined only once.						☐
	$\left(1-\frac{\text{\# variables defined more than once}}{\text{total \# variables}}\right)$						
	(4) Program segmentation.				☐		☐
	$\left(1-\frac{\text{maximum segment length}}{\text{total program length}}\right)$						
	(5) Data segmentation.				☐		☐
	$\left(1-\frac{\text{Amount of unused data}}{\text{total amount of data}}\right)$						
	(6) Dynamic memory management utilized.			☐		☐	
	(7) Data packing used.					☐	
	(8) Free of nonfunctional code.						☐
	$\left(1-\frac{\text{\# nonfunctional statements}}{\text{total \# statements}}\right)$						
	(9) no duplicate codes..				☐		☐
	$\left(1-\frac{\text{\# duplicate statements}}{\text{total \# statements}}\right)$						
	(10) Storage optimizing compiler/assembly language used.					☐	
	(11) Free of redundant data elements.						☐
	$\left(1-\frac{\text{\# redundant data elements}}{\text{\# data elements}}\right)$						
	MODULE METRIC VALUE = $\frac{\text{total score from applicable elements}}{\text{\# applicable elements}}$				☐		☐
	SYSTEM METRIC VALUE = $\frac{\text{sum of storage efficiency measures for each}}{\text{total \# modules}}$				☐		☐

while their impact on efficiency is felt during operation.

The iterative processing metric is based upon elemental scores for the following items:

. keeping loop independent computations outside of loops
. using an execution optimizing compiler
. avoiding recomputation of repeated expressions
. overlay usage
. avoiding bit/byte packing and unpacking in loops
. avoiding use of nonfunctional code
. coding decision statements efficiently
. overhead for module linkages
. overhead for operating system linkages

The data usage efficiency measure is based on the following criteria:

. grouping data for efficient processing
. initializing variables when declared
. avoiding mixed-mode expressions
. avoiding operations on uncommon units
. referencing and indexing data for efficient processing

The elements of the storage efficiency measure are as follows:

. allocating storage requirements to design
. using virtual storage facilities
. defining common data only once
. program segmentation
. avoiding unused data
. using dynamic memory management
. using data packing

. avoiding use of nonfunctional code

. avoiding duplicate statements

. using a storage optimizing compiler

. avoiding redundant data elements

The proposed efficiency measures incorporate virtually all of the maxims of efficient coding practice. By doing so, they hope to in some sense reflect the actual time and storage utilization of a computer program. Unfortunately, the metrics can only provide indications of possible efficiency or inefficiency. A program with an execution efficiency score of .93 will not necessarily run faster than one with a score of .85. This is due to two principal reasons. First, the metrics weigh all of the elemental scores equally, instead of providing greater weight to the factors contributing most to the program's time and storage utilization. But any such weighing scheme would need to vary from one program to another. Second, and more importantly, the executional efficiency of a computer program is a dynamic function which reflects the program's response to various inputs. The proposed measures are static in nature, and have no way to examine this behavior.

As an extreme example of this problem, one could take a program and add one redundant loop which is executed a large number of times. This would impact the running time of the program dramatically, but result in only a miniscule change in the program's efficiency score. Several lines of nonfunctional code would only slightly increase this measure since it is defined as

$$1 - \frac{\text{\# nonfunctional executable statements}}{\text{total \# executable statements}}$$

The impact is further diluted since this score is only one of nine elements contributing to the overall iterative processing metric.

Another difficulty with the proposed metrics concerns the fact that they are low level measures which deal primarily with the object code. Many of the measures examine program features which can be improved by today's optimizing compilers. These include taking loop invariant computations out of loops, factoring out recomputed subexpressions, rearranging decision statements to speed processing, performing array computations (referencing and indexing) efficiently, initializing variables, and flagging inaccessible statements and unreferenced data. (See [3] for a good discussion of optimizing compilers.) When such compilers are used, programmers should write source code in the clearest and most straight-forward manner, and let the compiler take care of improving it. It would be unfair to use metrics which do not account for this.

A number of serious questions regarding the applicability and meaning of the proposed efficiency metrics remain. If a program achieves a score of .86, say, what exactly does this mean? Should the job be recoded to make it more efficient, or is the score satisfactory? Should an alternative algorithm be used? If such an alternative is explored, is it meaningful to compare the metrics for the two programs?

Algorithmic complexity gets around these difficulties by dealing with time and storage directly, rather than with measures based on coding practices derived only indirectly from them. Time and storage are effective and consistent measures of system resource utilization. This means they can be used as yardsticks to compare different implementations of the same algorithm, competing algorithms, or even the different parts of a system. Thus, algorithmic complexity provides a framework within which both measures and techniques for improving software efficiency can be sought.

Unfortunately, the field of algorithmic complexity has not matured to the point where it is possible to predict the time and storage requirements of the typical computer programs written in actual practice. Furthermore, the state-of-the-art is such that a great deal of mathematical and computer science experience is necessary before an individual is capable of performing such analyses, and so the known techniques are not yet ready for widespread field use. In this sense, we are not currently able to propose concrete alternatives to the RADC efficiency metrics. Much of the research proposed and conducted in this effort is aimed toward narrowing this gap. In particular, the work on systematic analysis of algorithms is directed toward ultimately developing automated tools and aids for this purpose. Furthermore, the work on experimental analysis of algorithms provides a framework and methodology for ascertaining performance data to compare the relative efficiency of competing algorithms and to catalog such information for use by system designers and programmers.

SECTION 3

ANALYSIS OF ALGORITHMS AND COMPUTATIONAL COMPLEXITY

The research conducted in this effort falls into the area of computer science known as analysis of algorithms and computational complexity. The goal of work in this field is to quantitatively study the efficiency of algorithms for various computational tasks.

To a large extent, software is currently produced by first designing a solution to a problem, then coding it, and finally testing it to ascertain performance data. Algorithm theory focuses on the design phase of software production. The basic tenet is that software tasks often involve problems that can be formulated in an abstract, mathematical manner. Once a problem is suitably formulated, alternative methods for solving it can be studied, and the best available one selected for implementation. The advantages of this approach are that the performance of the selected algorithm can be estimated before coding actually begins, and design trade-offs can be considered more quantitatively.

The inherent computational difficulty, or complexity, of a problem is studied by developing lower bounds on the amounts of various computational resources, such as time and storage, required for its solution. This is usually done by deriving lower bounds on the number of operations or steps required for the solutions of problems such as matrix and polynomial calculations, sorting, or determining properties of graphs and other combinatorial problems. One example of a result of this type is that sorting n items requires at least log n! ≈ n log n comparisons between items.

Unfortunately, there are at present only a small number of techniques of generally limited power for deriving nontrivial lower bounds on the complexity of problems. Unless there exists a lower bound on the complexity of a problem which closely approximates the amount of resource used by the best known algorithm, the existence of a far more efficient method for solving the problem cannot be precluded. For example, it was long believed that the usual procedure for multiplying two general n x n matrices, using n^3 multiplications and n^3-n^2 additions, was optimal. However, a now-famous algorithm for this problem which uses only about order $n^{2.81}$ arithmetic operations was published by V. Strassen in 1969 [4]. A very recent algorithm by V. Y. Pan uses only order $n^{2.49}$ operations [5]. Despite these advances, the best lower bounds to date reveal only that order n^2 multiplications or divisions and n^2 additions or subtractions are required. Hence, either a better lower bound or an asymptotically superior matrix multiplication algorithm must exist.

In recent years, a number of significant advances have been made in the field of algorithms. These advances range from the development of faster algorithms for particular tasks, such as the fast Fourier transform, to the discovery of a certain class of important problems called "NP-complete" for which all known algorithms are computationally inefficient. The development of new algorithms that are better than those currently in use leads to both greater efficiency and the feasibility of solving larger problems. Conversely, knowing that a problem is characterized by a certain intrinsic degree of difficulty is significant since every program that solves the problem will have associated with it at least a certain minimal cost in terms of system resource utilization.

The Appendix to this document contains an outline of the major topics and issues addressed in the area of algorithm analysis and computational complexity. Also included there is an annotated select bibliography referencing materials which cover and survey most of the work which has been done to date.

In the remainder of this section, some shortcomings in current algorithmic complexity research are discussed. The aim is to point out areas where new approaches and more work are needed. Such constructive criticism of the field was helpful in identifying specific problems to be addressed as part of this study.

Algorithmic complexity research has tended to concentrate on the algorithms themselves, rather than with practical details relating to their eventual implementation. For this reason, the results of the research are often stated in terms of the asymptotic, or main dependence, as a function of the input size, n. As an example, if the running time of a particular algorithm is $c_3n^3+c_2n^2+c_1n+c_0$, it would be said to be of "order n^3", written $O(n^3)$. In such order-of-magnitude analyses, the multiplicative and additive constants (c_0, c_1, c_2, and c_3), which are dependent on the particular implementation of the algorithm, are not usually considered. Unfortunately, there has not been sufficient interest in establishing the relative sizes of these constants for various competing algorithms. An algorithm whose running time is $2n^3$ will be actually faster than one with running time $50n^2$ for $n < 25$, even though the asymptotic behavior of the latter is better. The point at which two such algorithms have equal running times is referred to as their "crossover point".

A typical scenario in the analysis of algorithm is an easy to understand initial algorithm with a running time of n^3, say, followed by more complicated algorithms with running times of $n^{2.5}$, $n^2 \log n$, n^2, etc. Zealous researchers seem to enjoy making contests out of such results. There is an unfortunate tendency among both researchers and practitioners to misinterpret their importance. While each successive algorithm may be asymptotically faster, generally speaking it is also more complex to understand and implement and involves more computational overhead (i.e., it has bigger constant factors). The asymptotically fastest algorithm is often best only for input sizes greater than those ever likely to be encountered in actual practice. Knowing just where the crossover points occur is essential if implementers are to choose the most efficient method for their particular circumstances.

Furthermore, many existing bounds have limited attention to the number of instances of some key operation used in the solution of a given problem (e.g., comparisons for sorting, multiplications or the total number of arithmetic operations for matrix product). While the overall running time of algorithms for the problem may be driven by such considerations, the effects of loop control and testing, memory accesses, and various bookkeeping chores should not be totally ignored. The costs o' these ancillary operations will determine the constant factors associated with an algorithm, and may constitute the primary reason for selecting one procedure over another.

Another shortcoming of algorithmic complexity research is that most investigations have focused on the worst case, rather than the typical, behavior of algorithms. The principal reason for this is that the worst case is usually far more tractable to mathematical analysis. Another difficulty is that there is often no way to identify reasonably the probability distribution of problem instances. (For example, a typical assumption for sorting is that all input permutations are equally likely.) Nonetheless, the average performance of an algorithm is probably far more important from a practical standpoint since the worst case may actually occur rarely, if ever.

One area where more useful work could be done is experimental studies of algorithm performance. (See, for example, reference [6].) In such investigations, alternative algorithms for the same task are implemented and their behaviors measured and compared using a standard input data base. Performance profiles, consisting of tables and graphs, showing execution times or storage requirements as a function of input size can be obtained from such experiments. This approach to algorithm analysis has the important advantage that real performance data on typical (viz., average case) inputs are obtained, instead of merely studying the algorithm in more abstract (mathematical) terms. When the actual crossover points become known as a result of such work, we strongly suspect that a few of the new, asymptotically fast algorithms will be found wanting for practical input ranges.

A catalog of performance profiles for the most important algorithms used to solve commonplace computational tasks, like those listed in the Appendix, would be a very powerful software design tool. This information could serve as a guide to software implementers in selecting the best algorithm for their particular operational environment. Similar performance data is available to system designers in other disciplines, but not to software engineers. For example, engineers consult tables giving the stress and strain properties of materials when designing a bridge. Based on such considerations, they choose a design which will meet the operational requirements (loading, weather, etc.). With software, we tend to build the system first and test it to see if it will withstand the load afterward. This is akin to erecting a bridge first and then driving traffic over it to see if it collapses!

While the general field of algorithm analysis has undergone enormous growth over the past decade, little attention has been given to developping systematic techniques for performing such analyses. Most of the work has been done by applying ad hoc procedures which are specific to the algorithms under consideration. A systematic study of this area is needed to better understand the underlying principles involved, and to identify and classify the major techniques used. This, in turn, would provide an engineering framework for the analysis of algorithms, making the techniques used more widely accessible and promoting their application as a software design methodology.

D. E. Knuth is regarded by most as the founder of this field. His three volume series (ultimately to contain seven volumes), The Art of Computer Programming [7], serves as both an encyclopedia of results and a model of the type of algorithm analysis which is needed. In a 1972 paper describing the goals of such research [8], Knuth states:

> "Analysis of algorithms is an interesting activity which contributes to our fundamental understanding of computer science. In this case, mathematics is being applied to computer problems, instead of applying computers to mathematical problems.
>
> Analysis of algorithms relies heavily on techniques of discrete mathematics, such as the manipulation of harmonic numbers, the solution of difference equations, and combinatorial enumeration theory. Most of these topics are not presently being taught in colleges and universities, but they should form a part of many computer scientists' education.
>
> Analysis of algorithms is beginning to take shape as a coherent discipline. Instead of using a different trick for each problem, there are some reasonably systematic techniques which are applied repeatedly. Furthermore, the analysis of one algorithm often applies to other algorithms.
>
> Many fascinating problems in this area are still waiting to be solved."

This statement holds equally true today.

SECTION 4

THE URI/RADC ALGORITHMIC COMPLEXITY EFFORT

To fulfill the objectives of the contract, given the context described above, several subtasks were identified and undertaken. These included the following:

1) surveying the state-of-the-art in several important problem areas,
2) critiquing the tenor and general direction of present research in algorithm analysis and computational complexity, and constructing an applied research plan for dealing with the perceived deficiencies,
3) developing systematic procedures for the analysis of algorithms, with the hope that the analysis methods developed might ultimately be automated,
4) developing techniques for coping with problems which are computationally intractable (i.e., not solvable without using a prohibitively large amount of computer time), and
5) designing and conducting experimental investigations of algorithm efficiency as a complementary, or sometimes alternative, approach to the other more theoretical work.

Specific technical problems were selected as a test bed for examining each of these issues.

4.1 Measures of Algorithmic Efficiency: An Overview

This document is an introduction to, and overview of, the entire effort. We began by describing the specific missions of this study, as delineated in the contract's Statement of Work. Next, we presented the historical context for this work in terms of prior RADC studies on software quality measurement. Attention was concentrated on the measures of efficiency which had been proposed. We discussed the relationship between efficiency and overall software quality, and gave a critical assessment of the metrics in terms of their ability to reflect actual time and storage utilization.

Then we gave an overview of the field of algorithm analysis and computational complexity, whose objective is to predict the execution behavior of computer programs. The Appendix contains further information in the form of an outline of the major topics addressed in this area. Also included there is a select annotated bibliography with references which cover and survey most of the work which has been done to date.

These perspectives provide the background for the particular research directions pursued in this study. The remainder of this section provides an introduction to, and a perspective on, the other parts of the series, which describe the results of these investigations.

4.2 The Performance of Algorithms: A Research Plan

One of the tasks to be undertaken as part of the Statement of Work for this contract effort was to develop a research plan for RADC in the area of algorithmic complexity. The second paper in the series constitutes that plan.

An important measure of the quality of a computer program is the amount of system resources required to execute it. At the level of analysis of the underlying algorithm, time and storage are standard and effective measures. Unfortunately, when a program is actually executed on a particular computer system, time and storage become much less precise measures of the quality of the program. This is because the executional behavior of a program is a complicated function of the efficiency of the underlying algorithm, the programming language used to implement the algorithm, the speed and architecture of the hardware, and features of the operating system.

Previous work in algorithmic complexity has focused attention almost exclusively on the time and storage requirements for particular computational problems (e.g., sorting, matrix multiplication). In this research plan, a more general approach to the issue is taken -- an examination of the performance of algorithms on actual computer systems. This plan recommends continuing theoretical work of an applied nature on important open questions in the field of algorithm analysis and computational complexity. The problem application areas considered include computational algebra, sorting and information retrieval, pattern matching in strings, combinatorial optimization problems, and computational geometry. Moreover, new and unresolved issues concerning the relationship between programming languages, computer architecture, and the performance of algorithms on computer systems are also identified.

The plan suggests a number of ways for advancing the state-of-the-art to solve problems in the area of algorithm performance. The questions raised are all of a practical nature, and the technology currently exists for addressing all of the issues discussed. In this sense, any one of the recommendations made could be regarded as a short-term task, although a systematic attack on all of the issues addressed would certainly constitute an ambitious long-range research plan.

4.3 Fast Computer Algebra

Another task to be performed under the Statement of Work was to survey previous work in the area of algorithmic complexity. To survey the entire field would have required an inordinate effort in view of the vast amount of research which has been undertaken over the past decade. In fact, several lengthy books have been written on the subject, none of which cover the entire field. Instead, we have provided an overview of, and pointers to, this work in the form of an outline and bibliography appearing as the Appendix to this document.

Additionally, we have selected two areas to survey in detail -- computational algebra and data base access methods. Furthermore, each of the papers in the series begins with a survey of work already done in the particular problem area it addresses.

The first of the surveys explores the topic of algebraic complexity. (See references [7b], [9], and [10] for more detail.) This problem area was selected because of its pervasive importance and because a good tutorial on the subject did not previously exist. The specific questions considered include the problems of raising some quantity (e.g., a number, polynomial, or matrix) to a power, evaluating a polynomial at one or several points, and multiplication of polynomials and matrices. Many new algorithms for solving such familiar algebraic problems on computers have recently been devised. These methods are more efficient than the classical ones for sufficiently large problem sizes, and some of them have now become quite famous (e.g., the Fast Fourier Transform, Strassen's matrix multiplication method).

In addition to surveying these most significant developments, the paper attempts to give a feeling for the spirit of how algorithmic complexity research proceeds. The nature and types of questions asked by researchers are explored. Several of the problems studied are shown to interact with each other in interesting, and perhaps unexpected, ways. General algorithmic design strategies, like divide-and-conquer, are applied to more than one of the problems. Some of the algorithms are shown to be optimal by deriving lower bounds.

4.4 Systematic Analysis of Algorithms

The Statement of Work also called for the development of systematic procedures for the analysis of algorithms. Virtually all of the work to date in analyzing algorithms has been done by applying ad hoc procedures. A growing number of such methods have been developed. but most of these are known to only a relatively small number of researchers. A systematic study of this area is needed to better understand the underlying principles, and to identify and classify the major techniques used. This, in turn, would provide an engineering foundation for the analysis of algorithms, making the techniques more widely accessible and promoting their application as a software design methodology.

Many current software engineering research efforts are aimed at automating software quality control investigations (e.g., program correctness analyzers, tools for measuring the "psychological complexity" of computer programs). Using techniques somewhat analogous to those employed in proving or checking proofs of program correctness, it should be possible to begin making progress toward the development of automated or semi-automated tools for symbolically analyzing the performance of programs or algorithms.

The gross limits of automatic algorithm analysis are known. The execution time of a computer program is not, in general, a decidable property. This follows directly from the well-known "halting problem" of computability theory (see [11], for example). Program correctness is also an undecidable property.

While any system which attempts to determine the correctness or execution time of an arbitrary program is doomed to failure, this does not imply that techniques cannot be developed to apply to a large class of the programs written in actual practice.

Cohen and Zuckerman [12] have built a prototype system called PL/EL which greatly aids in the analysis of algorithms. The system consists of a structured Algol-like language, called PL, for describing algorithms and an interactive command language, EL, for communicating and obtaining behavioral information. The programmer is required to specify the branching probabilities, and the system works out the details of the analysis using an algebraic manipulation package.

Wegbreit has developed both formal and prototype systems for analyzing program behavior. His prototype, called METRIC, is capable of analyzing simple LISP programs with less user-supplied information than PL/EL [13]. Wegbreit's formal system is based on Floyd-Hoare semantics, and the analysis of the algorithm is a natural by-product of formally verifying its correctness [14]. Recently, Ramshaw [15] has shown that there are some basic problems with Wegbreit's approach. He remedies these deficiencies by using frequencies, instead of probabilities, in his analyses. But due to the logical incompleteness of his axiom scheme, there are some simple programs which it cannot handle either. This flaw seems to be symptomatic of those formal systems of algorithm analysis which have grown from the work in program verification.

While program verification has not lived up to the promises which were held out for it several years ago [16], the field of automatic analysis is by no means played out. We have chosen a different route in the development of a formal system for algorithm analysis. Our approach is tied very closely to the semantics of a program. Looping is translated into equivalent recursive control structures so that recurrence relations describing program behavior are readily ascertained. Probability density functions are used to handle conditionals.

The approach is detailed in the fourth paper of this series. It has been advanced to the point where it can be successfully applied to a large class of problems, including all of the examples dealt with in previous work. The approach is systematic in that it treats all algorithms the same way. It holds promise for ultimately being automated. A necessary step in doing so would be to develop supporting techniques to solve the recurrence relations using an algebraic manipulation package (e.g., MACSYMA, REDUCE, MATHLAB). (See [17] for an elementary discussion of recurrence equations and techniques for their solution. Algebraic manipulation systems are discussed in [18].)

4.5 Adaptive Methods for Unknown Distributions in Distributive Partitioning Sorting

Experimental investigation of algorithm performance is one area we mentioned that needs more attention if the fruits of algorithmic complexity research are to be brought to bear as useful software design techniques. A catalog of performance profiles for important algorithms would be a welcome tool in assisting software designers to choose between alternative methods, and to estimate and understand the expected behavior of a system before it is implemented. Because we believe that this is such an important but neglected area, we selected it for one of the subtasks performed in this study. Our objective was to point out both the benefits of this type of analysis, as well as to explore issues of experimental design and develop a framework within which subsequent investigations might be conducted.

Timing statistics, themselves, are an inadequate measure of algorithmic performance since they are highly dependent on the machine and operating system used to run the experiments. Furthermore, in a paging or multi-programming environment, such statistics exhibit a large variance depending on system workload. Since we are interested primarily in comparing algorithms, rather than implementations, our approach is to count the number of times each straight-line section of code is executed when the algorithms are run on a large body of representative test data. Then weights are assigned to the various straight-line segments to reflect their relative costs in as implementation-independent a manner as possible. The experimental investigation described in the fifth part of this series served as as test bed to validate these ideas, using an exciting new sorting technique as a case study.

A large percentage of data processing applications is spent sorting data. For this reason, it is not surprising that sorting is perhaps the most widely studied problem in computer science. The faster data can be sorted, the more computer time and money can be saved. Most sorting methods are based upon comparisons between data items. Any such algorithm must use at least n log n comparisons (See the well-known lower bounding argument in Section 5.3.1 of reference [7c].) In 1978 W. Dobosiewicz, a Polish computer scientist, published a paper describing a new sorting technique called Distributive Partitioning Sorting (DPS) [19]. The method generated a lot of interest and excitement, and was considered by many to be a real breakthrough, because its expected running time was only O(n). This is possible because the procedure is not based upon item comparisons, like most conventional sorting algorithms in use (e.g., quicksort, heapsort, bubblesort), but upon ideas borrowed from distribution methods like radixsort.

Although DPS was an innovative method, a number of important problems remained before it could be considered a practical method for sorting data on a computer. First, the method was biased toward uniform data, performing poorly on skewed distributions. Second, the method incorporated several somewhat esoteric features which, although guaranteeing a linear expected running time, would result in very high constant factors and perhaps render the method of theoretical interest only. Third, the originally published version had several minor errors which were fairly easy to correct. Finally, DPS contained a number of places where some experimental fine-tuning could greatly improve the algorithm's performance. We undertook to remedy these deficiencies.

To show that DPS was indeed a practical sorting method, we benchmarked an implementation of it against quicksort. (Quicksort was developed in 1962, and is widely regarded as the fastest expected time sorting algorithm on most machines. See Section 5.2.2 of reference [7c] for more information.) We found that on uniform data, DPS performed better than quicksort for inputs of 750 or more items. We then developed two adaptations of DPS, called the Ranking Method and the Cumulative Distribution Function (CDF) Method, to deal with skewed data. These methods transform unknown distributions into uniform distributions and then perform the sorting.

Experiments were run on four algorithms (two versions of DPS, Ranking, and CDF) using four distributions (uniform, normal, Poisson, and exponential) for six input sizes (500, 1000, 5000, 10000, 20000, 30000 items). It was found that if it is known in advance that the data distribution will typically be uniform, normal, or only slightly skewed, then it is advisable to use DPS. However, if it is possible the data distribution might be very skewed, or if extremely large or small values exist relative to the rest of the data, then there is little to lose and a lot to gain by using the CDF adaptation. The CDF Method was only 2% to 4% slower than DPS in the uniform case, but ran up to 12% faster than DPS for 30000 items on exponentially distributed data. The Ranking Method was found to contain too much overhead to be competitive with DPS.

4.6 Expected Behavior of Approximation Algorithms for the Euclidean Traveling Salesman Problem

Combinatorial optimization has been one of the most actively studied application areas of algorithmic complexity research. This area encompasses a wide variety of problems such as finding properties of graphs and networks, optimal scheduling, bin packing, and set covering and partitioning. (See reference [20] for a thorough picture of this application area.) Despite the seeming diversity of these problems, similar algorithm design strategies can be used to solve most of them, and interesting relationships between many of the problems have been shown to exist.

The problems which have been studied in this area can fruitfully be divided into two categories, depending on their worst case execution times. The first class consists of those problems having algorithms whose running time is polynomial in the size of the input. Important examples include finding the shortest distance between two points in a network, the minimum spanning tree problem, maximizing flows in a network, matching and marriage problems, and testing a graph for planarity. To a novice, many of the problems which fall into this category at first seem computationally intractable, requiring essentially exhaustive enumeration procedures for their solution. Usually significant insights into a problem, exploiting some underlying structure, are required before polynomial time algorithms can be devised.

The second class of combinatorial problems are those for which no polynomial time algorithm is known. Algorithms for these problems generally resort to exhaustive enumeration of essentially all possible solutions, and in the worst case have exponential running times. A simple example is the subset sum problem. Here, we are given n positive integers $x_1,\ldots,x_n$ and another positive integer y. We are asked to identify the subset of x_i's whose sum is closest to, but does not exceed, the value of y. There are 2^n subsets of the x_i's, and it appears as though virtually all of these will have to be tested in the worst case. Other important combinatorial problems for which no polynomial time solution currently exists include 0-1 integer programming, the traveling salesman problem, testing for graph isomorphism (equivalence), minimal graph coloring, satisfiability of formulas in propositional logic, and a variety of covering, packing and partitioning problems on sets and graphs.

One might argue that the notion of polynomial time is too imprecise to be used as a criterion for classifying the computational difficulty of a problem. In fact, it has proven to be a very convenient measure. Actually, very few polynomial algorithms with running times of degree greater than 4 or 5 have ever been studied, although an $O(n^{100})$ procedure would unfortunately still meet this definition of "easy". Furthermore, Table 6 shows that even for moderate problem sizes (e.g., n=50), exponential algorithms are totally infeasible.

Most of the well-known combinatorial problems which appear to be intrinsically exponential belong to a class called the NP-complete problems, first explored by Cook [21] and Karp [22] and the subject of a recent book by Garey and Johnson [23]. The problems in this class are computationally equivalent in the sense that if a polynomial time algorithm is found for any one of the problems, then all of them can be solved in polynomial time. Results of this nature are obtained by constructing a polynomial time transformation mapping instances of one problem into equivalent instances of another. Conversely, if an exponential lower bound can be proven for any one of the problems in a sufficiently general model of computation, then all of the NP-complete problems will require exponential time. Most researchers in algorithmic complexity feel that this issue is the most important open queston in the entire field. Since this difficult question has been worked on by large numbers of prominent researchers, it appears that a satisfactory resolution may not be forthcoming for quite some time.

In view of the fact that instances of NP-complete problems arise frequently in actual computing practice, ways of coping with the apparent intractability of such problems must be devised.

Table 6

EXECUTION TIME VS. INPUT SIZE

TIME COMPLEXITY	INPUT SIZE N 2	10	50
N	0.002 SEC	0.010 SEC	0.050 SEC
N $\log_2$ N	0.002 SEC	0.033 SEC	0.282 SEC
N^2	0.004 SEC	0.1 SEC	2.5 SEC
N^3	0.008 SEC	1.0 SEC	125. SEC
2^N	0.004 SEC	1.024 SEC	35.7 CENTURIES
3^N	0.009 SEC	59.05 SEC	2.28×10^{11} CENTURIES
N!	0.002 SEC	60.48 MIN	9.64×10^{51} CENTURIES

1 OPERATION/MILLISECOND

One approach is to develop fast approximation algorithms, or heuristics, for their solution. Johnson [24] is a pioneering work in this area. Instead of looking for the optimal solution to an instance of a problem, these procedures seek to find acceptably good solutions, to within specified tolerances, but which operate quickly.

The Euclidean traveling salesman problem is perhaps the most famous example of an NP-complete problem. In this problem, a traveler has to visit each of a number of designated cities on a map and return home via the minimum distance route. All known algorithms which find the shortest tour have a running time which is exponential in the number of cities. In view of the computational infeasibility of finding this exact solution for even a moderate number of points, much attention has been focused on the quality of approximation algorithms for this problem.

Previous researchers have examined the ratio of the tour length produced by various heuristic methods to that of the optimal tour in the worst case. Rosenkrantz, Lewis, and Stearns [25] have considered several approximation schemes from this perspective. The best approximation algorithm to date for this problem has been developed by Christofides [26,27]. It has an $O(n^3)$ running time and is guaranteed to find a path whose length is within a factor of $1\frac{1}{2}$ times the optimum. Guarantees of this kind provide a warning about the possible dangers involved with using some particular method. However, such results tend to be overly conservative (pessimistic) since worst data seldom, if ever, is encountered in practice. Furthermore, there is experimental evidence that most reasonable approximation schemes perform about equally well on the average, although their worst case performances can vary greatly.

In the sixth part of this series, we consider the behavior of several approximations for the Euclidean traveling salesman problem. The expected length of the tour constructed by an algorithm is estimated from the order statistics of the distribution of the distance between points. (See, for example, reference [28] for an introduction to order statistics.) The approximation methods considered include nearest neighbor, arbitrary insert, nearest and cheapest insert, and two methods based on finding the minimal spanning tree (including Christofides' algorithm). For the distribution examined, all of the approximations are shown to produce a tour whose expected length is $O(\sqrt{n})$, where n is the number of cities, and at most a small constant factor (ranging from 25.7% to 87.5%) from optimal. These results show a marked improvement over the worst case bounds for the algorithms considered. In fact, the nearest neighbor and arbitrary insert methods are not known to produce a tour whose worst case performance ratio is bounded by any constant.

An important contribution of this work is to show how order statistics can be applied to say significant things about the expected behavior of heuristics for the Euclidean traveling salesman problem. There is no reason why these techniques could not be applied to approximation algorithms for other NP-complete computational problems, as well. To date, most research has focused on deriving worst case performance guarantees for these methods, while very little is known about their expected performance. Since many of these approximations can be characterized as "greedy" algorithms (i.e., they minimize or maximize some criterion at each step), they would make good candidates for the application of order statistics, provided it is possible to characterize reasonably the distribution of inputs. Further explorations of this type could prove to be most useful and interesting.

4.7 Data Base Access Methods

Data base systems is an area of computer science which has become increasingly important over the past few years. The reasons for this attention are immediately evident. Data base systems provide an enterprise with centralized control for its operational data. This contrasts with most enterprises today, where each application maintains its own private files. The advantages of centralized control include sharing of information among users, elimination of redundant data, ability to enforce certain standards on the way data is kept and handled, data integrity and elimination of data inconsistency, and maintenance of data security. References [29] and [30] provide an overview of current data base system technology.

Despite the enormous growth of activity in this area, a number of relatively unexplored and difficult problems remain. One question which has not been completely resolved deals with the selection of efficient storage structures and algorithms to access and update large data bases.

Data base algorithms are different from other algorithms in several significant respects. First, the most important measure of algorithmic efficiency is the number of input/output, or I/O, operations which must take place to execute an algorithm. On a typical computer system, the central processor operates at speeds about 1000 times faster than the external devices (generally disk drives) where the data base resides. Furthermore, data bases are dynamic in nature since information is continually being added, deleted, or modified. Thus, the focus is on the amount of I/O necessary to access and modify a collection of records in a data base, rather than on the amount of work done by the central processing unit, or CPU.

For these reasons, data base access methods was selected as the topic for a second survey undertaken as part of the contract effort. The seventh paper in this series presents a discussion of the efficiency of several strategies for accessing and maintaining large data files. The records in the file contain several fields, or variables. We want to be able to access the record(s) in the file with specified values of certain variables. The forms of the problem depend on the number of variables in a query, and whether these variables are specified by a single value or a range of values. (For example, an employee record might consist of name, social security number, department, position, and salary. We might want a list of all the employees in a certain department, or a list of all employees in that department earning a salary in some particular range.)

The storage structures surveyed include K-ary and radix trees which are utilized by the access methods presented, B-trees and extensible hashing for univariate access, and radix bit mapping and K-D-B trees for multivariate access. All of the techniques described are currently suitable for practical use.

4.8 An Experimental Evaluation of the Frame Memory Model of a Data Base Structure

One of the tasks undertaken as part of the contract effort was the evaluation of a storage structure model for data base systems.

A desirable goal of data base research is the automatic generation of data base structures. A designer would specify some limited number of characteristics of the data and would have automatically returned the data structures, the access items, and the access paths. A step in the direction of that goal would be for the designer to furnish usage information and a proposed storage structure, and to have returned the expected response parameters. The frame memory model of storage structure has been proposed as a mechanism for predicting system response as a function for usage and structural information. In this study, we report on an experimental validation effort for frame memory.

Most attempts at automatic design involve the following steps:

1. Determine how the users of the file system are planning to use the system. This provides the necessary input for the automatic design system. Usage is defined by the different types of records in the system, their lengths and fields, plus the expected frequencies of additions, deletions, modifications, and retrievals to records and subsets of records in the file.
2. Select a set of storage structures for the records based on usage patterns defined in step 1.
3. Evaluate how this set of storage structures perform in the anticipated environment. This evaluation must take into account the change that the storage structures will undergo due to maintenance.
4. Assign a rating to the set of storage structures based on this evaluation. This rating will determine whether or not the set of structures will be considered further as a possible design choice.
5. Inform the designer as to the set of structures which have received the best evaluations.

We are interested here in what is involved in step 3 of the design process. This step is complex partly because the amount of time needed to retrieve data from a storage structure rarely remains constant throughout the life of the storage structure.

March [31] has proposed that step 3 of the design process be divided into two steps as follows:

3a. Compute the average time to perform fundamental operations on the storage structure, taking into account the effects of updates to the storage structure. Fundamental operations include reading a logical block, scanning a logical block of records for a particular record, directly accessing a record, and writing a logical block.

3b. Use information from step 3a to calculate the average time to perform an operation of interest, which may involve a number of fundamental operations. For example, the operation of adding a record to a data structure can involve first the operation of reading in the logical block which will contain the record and then writing the updated logical block.

March proposed a model of secondary memory which he called frame memory. He also analyzed the cost of using this model to implement retrievals and modifications to a data base. The designer would specify data structure and retrieval requirements in terms of the frame memory. The cost of satisfying these requirements would be calculated and reported to the designer. The designer could then choose the best data structures.

This makes sense only if the equations used to predict the performance are correct and there is an implementation of frame memory so that the designer can then use this implementation to actually access the data structures created.

Under this contract effort, we undertook an implementation of frame memory. The implementation was tested to see if March's analysis yielded correct predictions. The results are described in detail in the eighth, and final, part of this series. They indicate that the predicted performance was close to the experimental values for almost all cases.

REFERENCES

[1] J. A. McCall, P. K. Richards, and G. F. Walters, "Factors in Software Quality", RADC Technical Report RADC-TR-77-369 (3 volumes), November 1977.

a. Volume I, "Concepts and Definitions of Software Quality". AD#A049014.
b. Volume II, "Metric Data Collection and Validation". AD049015
c. Volume III, "Preliminary Handbook on Software Quality for an Acquisition Manager". AD049055

[2] J. A. McCall and M. T. Matsumoto, RADC Technical Report RADC-TR-80-109 (2 volumes), April 1980.

a. Volume I, "Software Quality Metrics Enhancements". AD086985
b. Volume II, "Software Quality Measurement Manual". AD086986

[3] A. V. Aho and J. D. Ullman, Principles of Compiler Design. Addison-Wesley, 1977.

[4] V. Strassen, "Gaussian Elimination is Not Optimal", Numerische Mathematik, Vol. 13 (1977), pp. 354-356.

[5] V. Y. Pan, "New Combinations of Methods for the Acceleration of Matrix Multiplication", unpublished manuscript (1980).

[6] J. Cohen and M. Roth, "On the Implementation of Strassen's Fast Matrix Multiplication Algorithm", Acta Informatica, Vol. 6, No. 4 (August 1976), pp. 341-355.

[7] D. E. Knuth, The Art of Computer Programming. Addison-Wesley.

a. Volume 1, Fundamental Algorithms, 1968, 1973.
b. Volume 2, Seminumerical Algorithms, 1969, 1981.
c. Volume 3, Sorting and Searching, 1973.

[8] D. E. Knuth, "Mathematical Analysis of Algorithms", Proc. IFIP Congress '71, North-Holland (1972), pp. 19-27.

[9] A. Borodin and I. Munro, The Computational Complexity of Algebraic and Numeric Problems. American Elsevier, 1975.

[10] L. I. Kronsjö, Algorithms: Their Complexity and Efficiency. Wiley, 1979.

[11] M. L. Minsky, Computation: Finite and Infinite Machines. Prentice-Hall, 1967.

[12] J. Cohen and C. Zuckerman, "Two Languages for Estimating Program Efficiency", Communications of the ACM, Vol. 17, No. 6 (June 1974), pp. 301-308.

[13] B. Wegbreit, "Mechanical Program Analysis", Communications of the ACM, Vol. 18, No. 9 (September 1975), pp. 528-539.

[14] B. Wegbreit, "Verifying Program Peformance", Journal of the ACM, Vol. 23, No. 4 (October 1976), pp. 691-699.

[15] L. H. Ramshaw, "Formalyzing the Analysis of Algorithms", Stanford Univ., Ph.D. Thesis (June 1979).

[16] R. A. DeMillo, R. J. Lipton, and A. J. Perlis, "Social Processes and Proofs of Theorems and Programs", Communications of the ACM, Vol. 22, No. 5 (May 1979), pp. 271-280.

[17] G. S. Lueker, "Some Techniques for Solving Recurrences", ACM Computing Surveys, Vol. 12, No. 4 (December 1980), pp. 419-436.

[18] J. Moses, "Algebraic Simplification: A Guide for the Perplexed", Communications of the ACM, Vol. 14, No. 8 (August 1971), pp. 527-537.

[19] W. Dobosiewicz, "Sorting by Distributive Partitioning", Information Processing Letters, Vol. 7, No. 1 (January 1978), pp. 1-6.

[20] E. M. Reingold, J. Nievergelt, and N. Deo, Combinatorial Algorithms: Theory and Practice. Prentice-Hall, 1977.

[21] S. A. Cook, "The Complexity of Theorem Proving Procedures", Proc. 3rd Annual ACM Symposium on Theory of Computing (1971), pp. 151-158.

[22] R. M. Karp, "Reducibility Among Combinatorial Problems", in R. E. Miller and J. W. Thatcher (eds.), Complexity of Computer Computations, pp. 85-104. Plenum Press, 1972.

[23] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, 1979.

[24] D. S. Johnson, "Approximation Algorithms for Combinatorial Problems", Journal of Computer and System Sciences, Vol. 9, No. 3 (December 1974), pp. 256-278.

[25] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "An Analysis of Several Heuristics for the Traveling Salesman Problem", SIAM Journal on Computing, Vol. 6, No. 3 (September 1977), pp. 563-581.

[26] N. Christofides, "Worst Case Analysis of a New Heuristic for the Traveling Salesman Problem", Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon Univ. (1976).

[27] G. Cornuejols and G. L. Nemhauser, "Tight Bounds for Christofides' Traveling Salesman Heuristic", Mathematical Programming, Vol. 14, No. 1 (January 1978), pp. 116-121.

[28] E. J. Gumbel, Statistics of Extremes. Columbia Univ. Press, 1958.

[29] C. J. Date, An Introduction to Database Systems, Third Edition. Addison-Wesley, 1981.

[30] J. D. Ullman, Principles of Database Systems. Computer Science Press, 1980.

[31] S. T. March, "Models of Storage Structures and the Design of Database Records Based Upon a User Characterization", Ph.D. Thesis, Univ. of Minnesota (1978).

APPENDIX

Analysis of Algorithms and Computational Complexity

Outline and Bibliography

The goal of computational complexity is to quantitatively study the efficiency of algorithms for various tasks performed on a computer. Complexity theory investigates the inherent difficulty of a particular computational problem by deriving good lower bounds on the amounts of various resources, such as time and storage, required for its solution. This provides a framework within which the performances of alternative algorithms for the problem can be compared and improved methods of solution developed.

The major topics in this emerging discipline are listed in the outline which follows. The outline is divided into four principal sections:

1) general issues which delineate the scope of any particular algorithmic complexity investigation,
2) design strategies which have been used to develop new algorithms in diverse application areas and to categorize the problem-solving approaches embodied in most algorithms,
3) methods for deriving lower bounds, or theoretical minima, on the resource requirements to solve a given computational problem independently of the algorithms used, and

4) application areas which have been studied, together with the major problems which have been investigated within these areas.

For those interested in further information, an annotated select bibliography, listing the most important books and survey papers which have appeared, is also provided. Virtually all of the research published to date is accessible through references in the works cited.

Algorithm Analysis and Computational Complexity

An Outline

General Issues

1. Time and space analysis
2. Models of computation
   - Turing machines
   - computer-like models (RAMs and RASPs)
   - decision trees
   - straight-line programs (chains)
3. Exact vs. asymptotic analysis
   - measuring problem size
   - order-of-magnitude (O-, Ω-, Θ- notation)
4. Upper vs. lower bounds
5. Worst case vs. average case

Algorithm Design Techniques

1. Divide-and-conquer (recursion)
2. Greedy method
3. Dynamic programming
4. Basic search and traversal
5. Backtracking
6. Branch-and-bound
7. Approximation algorithms
8. Data structuring

Lower Bounding Methods

1. Trivial lower bounds
2. Decision trees
   - "information-theoretic" bounds
   - oracles and adversary arguments
3. Problem reduction/transformation
   - NP-completeness
4. Algebraic techniques
5. Miscellaneous tricks

Problem Areas

1. Ordering and information retrieval
   - sorting
   - merging
   - selection
   - searching
2. Algebraic and numerical problems
   - evaluation of powers
   - polynomial evaluation and interpolation
   - polynomial multiplication and division
   - matrix multiplication
   - greatest common divisors
   - factoring and primality testing

3. Graphs and networks
   - minimal spanning tree
   - shortest paths
   - connectedness and survivability (connectivity, transitive closure, articulation points, biconnectivity, strong connectivity)
   - circuits (Eulerian, Hamiltonian, traveling salesman problem)
   - graph coloring
   - network flows
   - planarity
   - isomorphism
   - cliques
   - bipartite matching

4. . Computational geometry
   - convex hull
   - closest point problems
   - intersection problems

5. Miscellaneous roblems
   - pattern matching in strings
   - cryptography
   - scheduling
   - operations research

Select Bibliography

Books

D. E. Knuth, The Art of Computer Programming (Vol. 1, Fundamental Algorithms; Vol. 2, Seminumerical Algorithms; Vol. 3, Sorting and Searching). Addison-Wesley, 1968, 1969, 1973.

Presents and discusses a wide spectrum of computational problems and algorithms. It is the authoritative source for algorithm theory, and does a nice job on certain aspects of complexity theory (e.g., the treatment of sorting, merging, and selection in Vol. 3). This classic work provides thoroughly comprehensive and historical coverage of its subject matter.

E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms. Computer Science Press, 1978.

A good one-volume introduction to the field. The book is organized around the major algorithm design techniques -- divide-and-conquer, the greedy method, dynamic programming, basic search and traversal techniques, backtracking, branch-and-bound, and algebraic simplification and transformations. Chapters on lower bound theory, NP-completeness, and approximation algorithms are also included.

A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

A more theoretically oriented one-volume overview of the field. Covers topics from a wide variety of problem areas. The book also formulates and compares several computer models such as random access register and stored program machines, and automata-theoretic models (e.g., Turing machines, finite automata, pushdown machines). Contains an outstanding bibliography.

A. Borodin and I. Munro, The Computational Complexity of Algebraic and Numeric Problems. American Elsevier, 1975.

An excellent monograph providing virtually complete coverage of its subject area. Considers such problems as polynomial evaluation, interpolation, and matrix multiplication.

E. M. Reingold, J. Nievergelt, and N. Deo, Combinatorial Algorithms: Theory and Practice. Prentice-Hall, 1977.

Discusses the complexity of a number of important combinatorial problems and analyzes the best known algorithms for their solution. Topics include exhaustive search techniques, generating combinatorial objects, fast sorting and searching, graph algorithms, and NP-hard and NP-complete problems.

M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completness. Freeman, 1979.

Detailed guide to the theory of NP-completeness. Shows how to recognize NP-complete problems and offers practical suggestions for dealing with them effectively. Provides an overview of alternative directions for further research, and contains an extensive list of NP-complete and NP-hard problems.

J. E. Savage, The Complexity of Computing. Wiley-Interscience, 1976.

Covers all of the significant results on the complexity of switching networks, and surveys several other problems in complexity theory. Importantly, this work also attempts to provide a framework for the quantitative study of time-storage tradeoffs and other performance evaluation criteria on models of real computers.

L. I. Kronsjö, Algorithms: Their Complexity and Efficiency. Wiley, 1979.

A mathematically oriented book. Its most important contribution is a detailed discussion of algorithms for numerical problems from the perspective of their numerical accuracy, as well as efficiency. Problems considered include polynomial evaluation, iterative processes, solving sets of linear equations, and the fast Fourier transform. Several nonnumerical applications, most notably sorting and searching, are also discussed.

S. Even, Algorithmic Combinatorics. Macmillan, 1973.

An early treatment of the basic questions explored in combinatorial mathematics. Algorithmic aspects of enumeration problems including generation of permutations and combinations, trees and their properties, and fundamental properties of graphs and networks are considered.

S. Even, Graph Algorithms. Computer Science Press, 1979.

A rigorous treatment of several applications and problems from graph theory. Trees and their properties, graph connectivity and searching, network flows, graph planarity, and NP-completeness are discussed in this monograph.

S. Baase, Computer Algorithms: Introduction to Design and Analysis. Addison-Wesley, 1978.

An upper-level undergraduate text covering selected topics from sorting, graphs, string matching, algebraic problems, relations, and NP-completeness. Aims to develop systematic principles and techniques for studying algorithms. Level of presentation is mathematically thorough.

S. E. Goodman and S T. Hedetniemi, Introduction to the Design and Analysis of Algorithms. McGraw-Hill, 1977.

An undergraduate text, oriented more toward students of programming and less mathematically rigorous than Baase. Like Horowitz and Sahni, this book is organized around the basic algorithm design methods, but its treatment is not nearly as comprehensive (usually one example per technique).

Survey Papers

B. Weide, "A survey of analysis techniques for discrete algorithms", Computing Surveys, Vol. 9, No. 4 (December 1977), pp. 291-313.

A good overview of the field. Discusses all the major issues including models of computation, measuring problem size and asymptotic complexity, lower bounding techniques, worst and average case behavior of algorithms, and approximation methods for NP-complete problems.

J. L. Bentley, "An introduction to algorithm design", Computer, Vol. 12, No. 2 (February 1979), pp. 66-78.

Another good introduction, written primarily for the novice. Contains more illustrative examples than Weide, but does not discuss issues in as much depth. Problems covered include subset testing (via sorting and searching), pattern matching in strings, the FFT, matrix multiplication, and public-key cryptography.

J. E. Hopcroft, "Complexity of computer computations", Proc. IFIP Congress '74, Vol. 3. (1974), pp. 620-626.

Discusses unifying principles in the design of efficient algorithms through the use of several well-chosen examples. More mathematically oriented than some of the other surveys.

E. M. Reingold, "Establishing lower bounds on algorithms -- a survey", AFIPS Spring Joint Computer Conf. '72, Vol. 40 (1972), pp. 471-481.

A clearly written survey of many of the early results concerned with deriving lower bounds on the complexity of functions. Emphasizes ordering (sorting, searching, merging, and selection) and algebraic problems.

R. E. Tarjan, "Complexity of combinatorial algorithms", SIAM Review, Vol. 20, No. 3 (July 1978), pp. 457-491.

Examines recent research into the complexity of combinatorial problems, focusing on the aims of the work, the mathematical tools used, and the important results. Topics covered include machine models and complexity measures, data structures, algorithm design techniques, and a discussion of ten tractable combinatorial problems.

R. M. Karp, "On the computational complexity of combinatorial problems", Networks, Vol. 5, No. 1 (January 1975), pp. 45-68.

A very readable introduction to the theory of NP-completeness.

Finally, the following articles in Scientific American provide a layman's introduction to most of the key issues in the field:

D. E. Knuth, "Algorithms", Vol. 236, No. 4 (April 1977), pp.63-80.

H. R. Lewis and C. H. Papadimitriou, "The efficiency of algorithms", Vol. 238, No. 1 (January 1978), pp. 96-109.

L. J. Stockmeyer and A. K. Chandra, "Intrinsically difficult problems", Vol. 240, No. 5 (May 1979), pp. 140-159.

N. Pippenger, "Complexity theory", Vol. 238, No. 6 (June 1978), pp. 114-124.

M. E. Hellman, "The mathematics of public-key cryptography", Vol. 241, No. 2 (August 1979), pp, 146-157.

R. L. Graham, "The combinatorial mathematics of scheduling", Vol. 238, No. 3 (March 1978), pp. 124-132.

R. G. Bland, "The allocation of resources by linear programming", Vol. 244, No. 6 (June 1981), pp. 126-144.

ALGORITHMIC COMPLEXITY
Part 2

Preface

An important measure of the quality of a computer program is the amount of system resources required to execute it. At the level of analysis of the underlying algorithm, time and storage are standard and effective measures. Unfortunately, when a program is actually executed on a particular computer system, time and storage become much less precise measures of the quality of the program. This is because the executional behavior of a program is a complicated function of the efficiency of the underlying algorithm, the programming language used to implement the algorithm, the efficiency of the code produced by the compiler, the speed and architecture of the hardware, and features of the operating system.

Previous work in algorithmic complexity has focused attention almost exclusively on the time and storage requirements of algorithms for particular computational problems (e.g., sorting, matrix multiplication). In this research plan, we take a more general approach to the issue — an examination of the performance of algorithms on actual computer systems. This plan recommends continuing theoretical work of an applied nature on important open questions in the areas of algorithm analysis and computational complexity. Moreover, new and unresolved issues concerning the relationships between programming languages, computer architecture, and the performance of algorithms on computer systems are also identified.

This plan, developed for Rome Air Development Center under Contract No. F30602-79-C-0124 (Algorithmic Complexity), suggests a number of ways for advancing the state-of-the-art to solve problems in the area of algorithm performance. The questions raised are all of a practical nature, and the technology currently exists for addressing all of the issues discussed. In this sense, any one of the recommendations made could be regarded as a short-term task, although a systematic attack on all of the issues addressed here would certainly constitute an ambitious long-range research plan.

CONTENTS

THE PERFORMANCE OF ALGORITHMS

A Research Plan

1. INTRODUCTION

1.1 Background

The notion of "algorithm" is of central importance in computer science and practice. An algorithm is a precise, step-by-step description of a computational procedure. To solve a problem on a computer, a human must communicate an algorithm for the problem to the machine using some language as a vehicle. This process is called programming, and the language used for communication is a programming language. In this view of computing, a program is merely the realization of an algorithm in a programming language.

Because algorithms play such a central role in computational processes, the performance of any computer-based system will depend to a large extent on the algorithms selected and how they are implemented in both software and hardware. In this research plan, we will explore three general approaches to this issue. The first of these is the classical study of algorithmic complexity, in which the time and storage resources required to implement a solution to a given problem are examined. Unfortunately, most research in this area usually stops at this point, although the impact of algorithms on the overall performance of computer systems extends to other levels.

Once an algorithm is selected, it must be coded in a particular programming language, and the features of the language chosen as well as its actual implementation will greatly affect system performance. Furthermore, the underlying hardware configuration of the machine on which the programs are to be executed will also impact the performance of the algorithms. The architecture of a computer system can affect the performance of algorithms in terms of both its suitability for the problem application area and its ability to efficiently support features of the programming languages to be used. Thus, the second and third approaches to be examined here are the impacts of programming languages and machine organization on the performance of algorithms.

To a large extent, computer-based systems are currently produced by first choosing a hardware configuration. Then, the traditional activities of the software development cycle occur. These activities include designing a solution to the problem, coding it, and finally testing the resultant implementation for correctness and to ascertain performance data. If a system does not meet the timing and storage utilization requirements in the specifications, portions of the programs are recoded. This often necessitates the use of assembly language, with its inherent difficulties in terms of writing, debugging, maintaining, and transporting software. Little thought is ever given to reexamining the algorithms used, and perhaps researching or developing alternative approaches. Programmers generally use only the standard algorithms with which they are familiar or, when faced with a novel application, use a direct "brute force" approach. Since the hardware configuration was frozen long ago, it is altogether too late to select an architecture which might have been more suitable to begin with.

Our approach to the performance of algorithms focuses attention on the overall design of a computer-based system. The basic tenet is that the tasks to be performed can generally be modeled in a precise (mathematical) or semi-rigorous way. Once a problem is suitably formulated, alternative algorithms for its solution can be studied and the best available one selected. The choice of algorithm can then aid in the selection of an appropriate programming language and machine configuration. The advantages of this approach are that the performance of the selected algorithm can be estimated before the coding and testing effort actually begins, and design tradeoffs can be considered more quantitatively.

1.2 An Overview

This research plan will be structured within two general frameworks. The first is a description of the three approaches to the performance of algorithms already identified:

1) algorithmic complexity,
2) programming languages, and
3) machine organization.

A discussion of how each of these issues directly impacts the performance of algorithms on actual computer systems is coupled with sugggested research topics aimed toward better understanding the underlying principles involved.

The field of computer science known as analysis of algorithms and computational complexity grew out of theoretical investigations of the inherent difficulty of solutions to programming problems in specific application areas. In the final portion of this paper, the most important of these areas are considered. These include the following:

1) computational algebra,
2) sorting, searching, and database systems,
3) pattern matching in strings,
4) combinatorial optimization problems, and
5) computational geometry.

The nature of previous work in each of these areas is discussed, and the most significant remaining issues which relate to the overall performance of computing systems are identified as areas worthy of further investigation.

Finally, an Appendix which puts previous research efforts on algorithm complexity into perspective is included at the end of the paper. This Appendix consists of an outline of the major topics, issues, and approaches which have been investigated, together with an annotated bibliography listing the key books and survey papers which have appeared.

2. APPROACHES TO THE PROBLEM

2.1 Algorithmic Complexity

Research into algorithmic complexity concentrates on the algorithm itself, rather than its implementation in any particular programming language or on any given machine. This type of research is predicated on the assumption that there are things about an algorithm which are true regardless of its implementation. For this reason, the results of this research are often stated in terms of the asymptotic behavior, or main dependence, as a function of the input size, n. As an example, if the running time of a particular algorithm is $c_3n^3+c_2n^2+c_1n+c_0$, it would be said to be of "order n^3", written $O(n^3)$. In such order-of-magnitude analyses, the multiplicative and additive constants (c_0, c_1, c_2, and c_3), which are dependent on the particular implementation of the algorithm, are not usually considered. Fortunately, there is a growing interest in establishing the relative sizes of these constants for various competing algorithms. An algorithm whose running time is $2n^3$ will be faster than one with running time $50n^2$ for $n<25$, even though the asymptotic behavior of the latter is better. The point at which two such algorithms have equal running times is referred to as their "crossover point".

Both theoretical and experimental research are possible in this area. The theoretical investigations include the determination of upper and lower bounds on the behavior of algorithms for a particular problem, the systematic analysis of algorithms, and the investigation of computational tradeoffs (e.g., between time and storage). Experimental investigations concentrate on measuring the typical behavior of complex algorithms over a wide range of input sizes. The object is to determine the crossover points of alternative algorithms which perform the same function. This is equivalent to determining the relative sizes of the additive and multiplicative constants which specify the behavior of the algorithms under study.

Much of the classical work in algorithmic complexity has been directed toward bounding the number of operations required to perform various computational tasks. For example, a well-known result of this kind states that at least $\log_2 n! \approx n \log_2 n$ comparisons are required to correctly sort a list of n items. This number is called a lower bound, and holds irrespective of the method used. Any comparison-based sorting algorithm may use more comparisons, but it cannot employ fewer. When we look at any particular sorting algorithm and determine the number of comparisons it uses, then we find an "upper bound" on sorting. The known algorithm using the fewest comparisons establishes the best upper bound to date. The object of a good deal of algorithmic complexity research is aimed at bringing the theoretical lower bound and practical upper bound together for various problems.

Upper and lower bounds provide a convenient yardstick for assessing the relative efficiency of an algorithm. As new algorithms and problems are investigated, this approach will doubtlessly continue to prove fruitful. There are, however, several areas where more work should be done to improve the usefulness of such results.

1) Existing bounds limit attention to some key operation associated with the solution of a given problem (e.g., comparisons for sorting, multiplications or the total number of arithmetic operations for matrix product). While the overall running time of algorithms for the problem may be driven by such considerations, the effects of loop control and testing, memory accesses, and various bookkeeping chores should not be totally ignored. The costs of these ancillary operations will determine the constant factors associated with an algorithm, and may constitute the primary reason for selecting one procedure over another.

2) Most bounding investigations have focused on the worst case, rather than typical, behavior of algorithms. The principal reason for this is that the worst case is usually far more tractable to mathematical analysis. Another difficulty is that there is often no way to reasonably identify the probability distribution of problem instances. (For example, a typical assumption for sorting is that all input permutations are equally likely.) Nonetheless, the average performance of an algorithm is probably far more important from a practical standpoint since the worst case may actually occur rarely, if ever.

3) There are at present only a small number of techniques of generally limited power for deriving lower bounds on the complexity of functions which are nonlinear in both the number of inputs and outputs. Unless there exists a lower bound on the complexity of a function which closely approximates the amount of resource used by the best known algorithm, the existence of a more efficient procedure cannot be precluded. Unfortunately, interesting lower bounds are quite difficult to derive mathematically, and we may have to be satisfied for some time to come with our intuition concerning the optimality of certain algorithms.

One technique of research seems to have been overlooked in this area. This is to produce a systematic catalog of algorithms, arranged by function, giving their known or suspected upper and lower bounds. This type of listing has proved to be a useful technique in the past to help identify the underlying order in the objects being studied. Making a catalog of important algorithms is different from the way that computer science has been done so far. Even if this kind of project does not result in any new discoveries, it will be of considerable value to those who are interested in the engineering aspects of algorithms.

Many current software engineering research efforts are aimed at automating software quality control investigations (e.g., program correctness analyzers, tools for measuring the "psychological complexity" of computer programs). Using techniques somewhat analogous to those employed in proving or checking proofs of program correctness, it should be possible to begin making progress toward the development of automated or semi-automated tools for symbolically analyzing the performance of programs or algorithms.

The gross limits of automatic algorithm analysis are known. Wegbreit [61] has constructed a system which can analyze simple LISP programs automatically. No completely automatic system or complete formal system can be constructed which can analyze all programs. This is firmly established by computability theory. In between the simple programs and "all possible programs", there is a lot of ground which can be covered.

Cohen and Zuckerman [18] have built a system which greatly aids in the analysis of algorithms. Their system helps the analyst with the details of the analysis while requiring the analyst to provide the branching probabilities. Wegbreit [62] developed a formal system for the verification of program performance. His technique can also be used to provide the branching probabilities which are needed. Recently, Ramshaw [50] has shown that there are problems with Wegbreit's probabilistic approach and has developed a formal system which he calls the frequency system.

While Ramshaw's frequency system can handle some of the programs that Wegbreit's cannot, there are some simple ones which it cannot handle either. In particular, it cannot handle the "Useless Test":

```
    if C then nothing else nothing endif
```

As Ramshaw points out, "The incompleteness of our Conditional Rule has its roots in one of the basic choices behind the frequency system: that assertions should specify sets of frequentistic states." This seems to be symptomatic of those formal systems of algorithm analysis which have grown from the work in program verification.

While program verification has not lived up to the promises which were held out for it several years ago [23], the field of automatic analysis is by no means played out. Anderson and Lamagna [2] have chosen a different route in the development of a formal system of algorithm analysis. Their approach is tied very closely to the semantics of a program. Looping is translated into equivalent recursive control structures so that recurrence relations describing program behavior are readily ascertained. Probability density functions are used to handle conditionals. The approach has been advanced to the point where it can handle the "Useless Test", as well as all the other programs covered by Ramshaw in his thesis. This work is encouraging, and further improvements should be possible.

Important work on mathematical symbolic manipulation programs (e.g., MACSYMA [45], REDUCE [31], MATHLAB [26]), directed toward developing techniques for automatically solving recurrence equations, will also be required to support the automatic analysis of algorithms. Work on this problem can proceed independently of that on formal techniques for analyzing algorithms. However, a final prototype will be easier to build if the interfaces between the parts are carefully defined in the beginning.

Another aspect of algorithmic complexity in need of further investigation is that of computational tradeoffs between important system parameters, such as time and storage. The development of a framework within which such tradeoffs could be quantitatively studied would be of considerable importance in the overall design of computer systems. It would facilitate comparisons between alternative design strategies, and would enable the estimation of performance parameters prior to implementation and testing.

Savage [52] has been able to derive two computational inequalities involving the time and storage required to compute a function. These inequalities provide a lower bound on the best performance one can hope to achieve when carrying out a given computational task. The framework used to develop these inequalities results from the juxtaposition of several subjects in theoretical computer science (e.g., switching and automata theory) with mathematical modeling of general-purpose computers and their associated storage devices.

Savage's first inequality states that in order to compute a given function f on a general-purpose computer M

$$C(f) \leq k_1 ST$$

where

C(f) is the minimal size combinational switching network which computes f directly,

S is the storage capacity of the machine M,

T is the number of cycles (i.e., time) used to compute f on M, and

k_1 is a positive constant.

The second inequality states that

$$I(f) \leq k_2(S+Tb)$$

where

I(f) is the minimum amount of information (i.e., minimal sized program) which must be supplied to compute f,

S is the amount of information that M has initially,

Tb is the amount of information which can be supplied to M in T cycles using input words of b bits in size, and

k_2 is a positive constant.

These inequalities define bounds on the time-space tradeoffs which can be achieved on real computing machinery (see Figure 1). Furthermore, they suggest using computational cost measures of the form αST and $\beta S+\gamma T$, which for nonnegative constants, α, β, and γ, are minimized when S is small and T is large, or vice versa. This tends to support the cost effectiveness of minicomputers and multiprogramming.

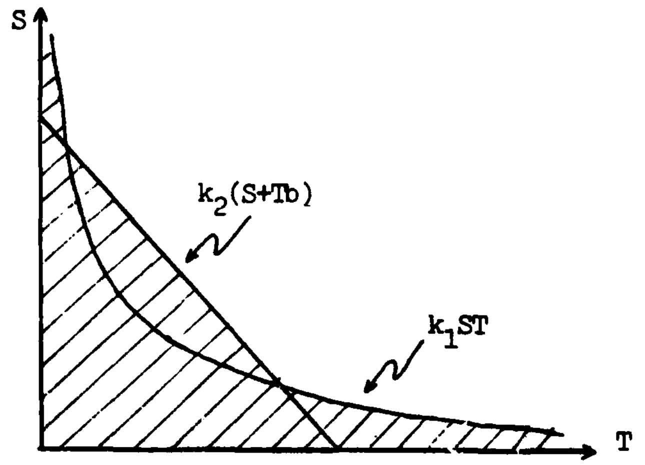

Figure 1. Storage-time boundaries. Forbidden region appears shaded.

Although the approach described above provides a general framework in which time-storage tradeoffs can be studied, little is known about the time-storage curves of particular functions. Recently, however, the pebble game on directed acyclic graphs has been used to model the space-time behavior of straight-line algorithms. Using this paradigm, nontrivial lower bounds on the product of storage and time have been obtained for such problems as the FFT [53], polynomial multiplication [59], matrix multiplication and inversion [34], sorting [8], transitive closure [60], and the class of linear recursive programs [54]. More work of this nature should be encouraged since the results could be extremely helpful in designing computer software. By matching available resources against a time-storage curve for the problem application at hand, one could be confident in choosing the most appropriate design strategy.

Investigations of other types of computational tradeoffs are also possible. It is well-known that in information retrieval there is a tradeoff between the time required to insert or delete an item in a data structure and the time required to search for an item. For example, insertion or deletion is a very simple operation in a linked list, but the worst case and average case search times for this structure are both proportional to the number of items in the list. At the other extreme, information can be retrieved from a perfectly balanced binary search tree in time proportrional to the logarithm of the number of items in the structure, but the insertion or deletion time is proportional to the total number of items in the tree if one insists that perfect balance be maintained. There are, of course, other storage strategies which have been developed to achieve a desirable compromise between insertion/deletion time and retrieval time. Although this issue has been studied with respect to particular data structures, there is no underlying general rule stating the fundamental limits within which such tradeoffs can be achieved.

One final area where much useful work could be done is experimental measurements of algorithmic performance. As mentioned earlier, theoretical work on the analysis of algorithms has focused almost exclusively on order-of-magnitude, or "big-oh" notation, results. Such results must be used with a note of caution. A frequent scenario in algorithm design is the development over time of asymptotically better procedures for some problem through the use of increasingly sophicticated data structures and clever solution strategies. Oftentimes, as the asymptotic behavior of the methods improve, the constant factors rise correspondingly. Theoreticians, in their exuberance at finding asymptotically better algorithms, have tended to overlook the constant factors, and the implications of their results have consequently been often misinterpreted by practitioners.

Few experimental studies have been performed where alternative algorithms for the same task have been implemented and their behaviors measured and compared using a standard input database. Performance profiles, consisting of tables and graphs, showing execution time or storage requirements as a function of input size could be obtained from such experiments. These measurements would serve as a guide to software implementers in selecting the best algorithm for their operational environment. This approach to algorithm analysis has the important advantage that real performance data on typical (viz., average case) inputs are used, instead of merely studying the algorithm in more abstract (theoretical) terms. When the actual crossover points become known as a result of such work, we strongly suspect that a few of the new, asymptotically fast algorithms will be found wanting for practical input ranges.

A catalog of performance profiles for the most important algorithms used to solve commonplace computational tasks would be a very powerful software design tool. Similar performance data is available to system designers in other disciplines, but not to software engineers. The problem application areas listed in the outline in this paper's Appendix could serve as a convenient starting point for collecting data. Because of the large number of algorithms and problem areas which have been systematically studied, this project is an ideal candidate for "distributed research". If such a modular approach is taken, however, it would be wise to set some uniform guidelines at the outset so that the results of the separate data collection efforts would all be on a comparable footing. Care must be taken in performing the experiments to minimize the effects of differing processor speeds and organizations, the compilers used and the degrees of optimization which they perform, and operating system differences. Performance data from multiprogramming and timesharing environments may also be subject to fluctuations in system workload.

2.2 Programming Languages

A program in a computer language is an interface between an abstract specification of an algorithm and the implementation of that algorithm on a computer. As such, there are two issues related to the performace of this interface. These are:

1) Human performance. How best to insure easy implementation of the algorithm, correctness of the program, portability, and maintainability.
2) Machine performance. How does the algorithm, as implemented in a programming language, perform on a computer?

Most work in this area has been focused on human performance. Historically, this was manifested by the development of higher level programming languages -- both general purpose (e.g. FORTRAN, COBOL, PL/I, more recently PASCAL, ADA) and application oriented languages (e.g. SNOBOL, APL, LISP, GASP). Much of the recent work on the human performance issue has been in the area called software science originally developed by Halstead [30]. Software science has evolved into an area of its own and represents an important line of continuing research.

The study of the machine performance of a high-level language implementation of an algorithm, as opposed to the performance of an algorithm itself, has proceeded in two directions. Initially, this took the form of attempting to correct for the inefficiencies inherent in using high-level languages and implementing optimization algorithms within compilers to generate more efficient machine code.

A recent article by Oldehoeft and Bass [48] provides a mechanism for pursing other dimensions of the machine performance of high-level language implementations of algorithms. This is the notion of counting data movements (work), both explicit and implied, within a program. Explicit data movements are those in direct response to statements and operators in the language. Implicit data movements are those that are done to prepare for the explicit data movements. A comparison of the explicit work (language work) of an algorithm implemented in two different programming languages gives a measure of the execution time appropriateness of those languages for that algorithm.

One of the issues that needs to be examined in the area of programming languages involves the tradeoff between machine performance and human performance. Some specific tradeoffs that should be examined are:

1) What is the machine performance loss due to the use of recursion? Recursion provides a powerful tool which simplifies the coding of many algorithms but, unfortunately, the resultant programs are usually characterized by poorer time and storage performance than purely iterative ones. This is due to the fact that the general

implementation of a recursive procedure call requires a run-time stack whose size is proportional to the number of levels of recursion. Additionally, the procedure calling mechanism itself takes more time to execute than a nonrecursive one. A determination of the costs involved will enable a decision to be made in particular cases about the need for human efficiency as opposed to the need for machine efficiency.

Some recursive algorithms are known to have relatively simple and far more efficient iterative realizations (e.g., the factorial function), while others do not (e.g., an iterative version of quicksort essentially duplicates the stacking mechanism inherent in the recursive version). A promising line of research would be to characterize the class of recursive algorithms which can be simply translated into iterative programs without in effect simulating the implementations of a general recursive procedure call. Such work would be important since optimizimg compilers might be designed to perform this conversion automatically. This would combine the advantages of allowing the programmer the convenience of recursion, while maintaining an efficient run-time environment.

2) What is the increase in psychological complexity for a more efficient algorithm for particular problems? A typical scenario in the analysis of algorithms is an initial easy to understand algorithm which has running time n^3, followed by more complicated algorithms with running times $n^{2.5}$, $n^2 \log n$, n^2, etc. Each successive algorithm is asymptotically faster but, generally speaking, is more complex and involves more initial overhead. The problem then becomes, for a particular algorithm or class of algorithms, at what point is the complicated formulation and initial overhead not worth the effort.

2.3 Machine Organization

The performance of any algorithm will ultimately be limited by the characteristics of the computer on which a programmed version of the algorithm is executed. Clearly, the same algorithm, expressed in the same programming language, will run several orders of magnitude faster on one of today's large scale scientific machines (e.g., Cray-1 or IBM 370/168) than on a microcomputer (e.g., an Intel 8080-based system). Computer hardware affects the performance of algorithms in three principal ways:

1) at the logical level (i.e., the technology and methods used to implement the underlying adders, shifters, comparators, decoders, etc.; the processor and memory cycle times),
2) at the level of processor organization (i.e., uniprocessors, pipelined architectures, parallel and distributed logic), and
3) at the level of the suitability of the machine architecture for the application at hand.

The functions performed by the circuitry of a computer can themselves be analyzed from an algorithmic perspective. Two important measures of the complexity of a combinational switching circuit are its size (the number of logic elements) and its depth (the number of levels of logic). Size is directly related to the cost of building the circuit and has an important effect on reliability. The more circuit elements there are, the more likely one will malfunction and the entire unit fail. A circuit's depth determines the delay inherent in its use. A circuit of depth d built from logical elements with delay t requires time dt to operate. Savage studies these two measures and their interrelation in [52].

The following example illustrates how the algorithms used to implement a computer's arithmetic and logical functions can dramatically impact the execution times of programs running on the machine. The standard method for adding two numbers of width w (i.e., w is the number of digits or bits) operates in time proportional to w. The conditional-sum algorithm, a highly parallel method, operates in time proportional to $\log_2 w$. For w=32, the speed-up ratio is 32/5=6.4. A similar improvement can be realized in most other hardware functions. The effect is even more dramatic for a multiply instruction. The classical method, which employs successive shifting and adding, operates in time w^2 while a parallel method operating in time $(\log_2 w)^2$ is known. For n=32, the speed-up is $(32)^2/(15)^2 \approx 41$. See Savage [52] for a catalog of such results.

Kuck [41] and Savage [52] have both developed frameworks within which the structure of computers and their computations can be analyzed. The methods they espouse can be used as systematic tools for examining alternative processor designs to estimate their performance characteristics. Applied research to test the validity of these approaches and to make them more widely available as practical design techniques would be an important contribution.

Some basic research into the complexity of logical functions may also be worthwhile. Pattern matching and string processing is an area which has received little attention in the past because these functions are seldom performed directly in hardware. Winograd [64] and others have proposed schemes with minimal delay for arithmetic functions like addition and multiplication, but the methods rely on encodings of the numbers which have never been used. A study of the practicality of incorporating such schemes into real computer systems could also prove interesting. Finally, the complexity of switching circuits can be used as a springboard for investigating the complexity of functions normally performed in software. Lamagna's study of monotone networks for sorting and merging, bilinear forms, and routing problems is an example of such work [42].

Most of the algorithms studied to date have been designed for single processor computers. As a result, the algorithms generally operate on data in a sequential fashion. Over the past decade, supercomputers with pipelined CPUs (e.g., Texas Instruments Advanced Scientific Computer [58], Control Data STAR-100 [19], CRAY-1 [21]) and multiple processors (e.g., ILLIAC IV [4]) have been designed and built to increase computational speeds and throughput. An algorithm which is optimal for a single processor, may not be anywhere near optimal in the environment of a parallel machine organization. Some research has been done on parallel algorithms for various computational problems (see [40] for a survey), but in much of this work the number of processors is unrealistically assumed to be unbounded. If full advantage is to be taken of the newer processor organizations, algorithms which more fully exploit the parallelism in these architectures should be developed and evaluated.

An issue related to the implementation of algorithms on computers with multiple CPUs is that of programming languages and language constructs for parallel processing. Present day programming languages, even those containing concurrency primitives, are woefully inadequate for this task. In view of the difficulties that programmers are likely to have when thinking in parallel terms to control efficiently multiple cooperating computations, much attention should be given to this question. Many of the problems encountered in producing quality software before the widespread acceptance of structured programming will probably reoccur if parallel algorithms are coded using undisciplined language constructs. Because of a likely increase in the importance and application of parallel processing techniques over the next decade, we anticipate that much careful study will have to be devoted to this problem.

Today, high-level programming languages are preferred to assembly languages for most programming applications. This is because it is far easier to express algorithms in high-level languages, which are user-oriented, rather than assembly languages, which mirror machine languages and are concerned exclusively with controlling hardware features of computing equipment. This fact has many important ramifications including the greater degree of readability, correctness, maintainability, and upgradability of code written in high-level languages. It is often the case that the higher level language constructs which facilitate the coding of algorithms must be translated into machine language representations which are inefficient in terms of execution time and memory space. This occurs because the classical von Neumann-type computer architecture employed by virtually all machines operating today is concerned primarily with the word-at-a-time flow of information between CPU and memory, rather than with actual problem-solving.

Von Neumann computers provide a single basic architecture for all applications. The motivation which led to the development of new programming languages to facilitate the description of algorithms in different application areas should serve as a model for developing new architectures to facilitate the execution of algorithms expressed in these languages. Unfortunately, there is often little or no interaction between computer designers and "software people". Although recent advances in hardware technology (e.g., large scale integration, pipelining, etc.) have been dramatic, the instruction repetoires of today's computers are very similar to those of their predecessors of 10 to 20 years.

Computer architecture can be used as a vehicle to improve the executional efficiency of high-level language programs. Machine instruction sets similar to the intermediate code generated by a compiler (e.g., quadruples, triples) provide a means for improving storage requirements and execution time when compared to the conventional method of machine code generation and execution. Carlson [12] surveys much of the previous work in designing high-level language computer architectures. Due to rapidly advancing hardware technology and recent advances in microprogramming techniques, the implementation of such machines has become both technologically and economically feasible. Microprogramming, in particular, provides a flexible and effective tool for engineering new classes of computers [1].

More work is needed in the area of designing, building or microprogramming, and experimentally testing high-level language computer architectures. Particular attention should be paid to languages with special features which greatly facilitate the specification of algorithms. Prime candidates are APL [33] and SNOBOL [29].

APL contains a rich set of primitives which allow the programmer to specify complex vector manipulation algorithms very concisely. The language is better suited to processing arrays of data items than scalar-oriented languages like FORTRAN and ALGOL. Because of the inherent parallelism in these array operations, a hardware or firmware implementation of the language should perform substantially better than a software implementation on a sequential von Neumann architecture. Furthermore, because the language is interactive and allows dynamic data types, a number of attribute binding and type and subscript checking operations must be deferred until execution time. This run-time flexibility increases the desirability of a microprogrammed implementation.

SNOBOL4 is a convenient language for succinctly expressing algorithms for pattern matching and the manipulation of character string information. Because the features of the language are far removed from those of von Neumann-type machines, elaborate software routines are required to implement the language on conventional computers. This implies that huge amounts of memory are generally required and program execution is quite slow. Thus, the features provided in the language are prime candidates for investigating the relationship between classes of algorithms and their efficient execution in hardware or firmware.

3. PROBLEM APPLICATION AREAS

3.1 Computational Algebra

Computational algebra is the study of algorithms for numerical applications and to manipulate mathematical formulas. Typical problems falling in this area include raising a number to a power, polynomial evaluation and arithmetic, and matrix manipulations. Although this is perhaps the oldest and most studied application area in algorithm theory, a good introduction did not exist until recently. Lamagna discusses the problems mentioned above from an algorithmic complexity perspective in a tutorial paper [43]. This much needed work surveys the major results and open questions, discusses the interplay between problems, and gives examples of the most widely used algorithm design techniques.

Computational algebra is an area where experimental research on the analysis of algorithms would be quite beneficial. Virtually all of the work to date on analyzing and comparing algebraic and numerical algorithms has been

AD-A118 813   RHODE ISLAND UNIV KINGSTON DEPT OF COMPUTER SCIENCE --ETC   F/G 9/2
ALGORITHMIC COMPLEXITY. VOLUME I.(U)
JUN 82   E A LAMAGNA, L J BASS, L A ANDERSON   F30602-79-C-0124
UNCLASSIFIED   81-161-VOL-1   RADC-TR-82-152-VOL-1   NL

2 of 2

END
DATE
FILMED
9 82
DTIC

done in a theoretical setting. Researchers have counted the number of scalar arithmetic operations (e.g., additions, multiplications, etc.) that various procedures utilize, but have ignored the cost of the other operations necessary to actually program the algorithms to run on computers. This overhead includes loop control and testing operations, as well as the time required to access information stored in the computer's memory. Furthermore, most published algorithm analyses are stated in terms of orders of magnitude, and ignore constant factors of proportionality.

Cohen and Roth [17] have compared Strassen's algorithm with the classical method for multiplying square n x n matrices [57]. "In theory", Strassen's algorithm is superior since it uses a number of arithmetic operations growing as $n^{2.81}$ to the classical algorithm's n^3. However, Cohen and Roth found experimentally that the classical algorithm was faster for matrices of size n less than about 40. They also found a straight-forward recursive implementation of Strassen's algorithm to have excessive overhead, and were driven to custom-tailor a more efficient version. More experimental work of this kind is sorely needed if the many new results of algorithm theory over the past decade are to benefit actual programming practice. Alternative algorithms for many basic algebraic and numeric problems would nicely lend themselves to this type of comparison.

Programming languages and the notion of "language work" (see Oldehoeft and Bass [3]) play a significant, yet unexplored, role in the implementation of algebraic algorithms. Few would disagree that FORTRAN is clearly better for programming numerical procedures than COBOL. But a language like Iverson's APL [33] is even more suitable for applications in this problem domain. The language supports features which enable users to succinctly specify complex vector and array manipulation algorithms, which form the heart of most problems in computational algebra. APL's power derives from two sources: (1) its dynamic features for specifying the shapes and types of data, and (2) the elegant way that a few general operators (e.g., inner product, outer product, reduction) can be combined to perform a wide range of functions.

John Backus, in his 1977 ACM Turing Award Lecture, goes so far as to credit APL for being the first language not based on the lambda calculus which is free from the primitive word-at-a-time style of programming inherited from von Neumann computers [3]. It is this freedom from conventional programming structures which gives APL its expressive power. Even though APL may free the programmer from thinking in word-at-a-time terms, the performance of algorithms written in the language must still suffer from what Backus calls the "von Neumann bottleneck" for implementations of the language on conventional computers. This bottleneck stems from the fact that only a single word at a time can be transmitted between the central processing unit (CPU) and the memory store. The task of a program is to alter the contents of the store in some significant way. Since conventional computers accomplish this change by shuttling vast amounts of information, both data and instructions, between the CPU and memory, we have grown accustomed to a style of programming that largely concerns itself with traffic through the bottleneck rather than with the larger conceptual units of our problems.

Here again, more research is needed on the relationship between computer hardware and algorithms. Conventional computers have, in general, been designed around instructions sets which support mainly arithmetic operations rather than, say, character manipulation or list processing. But, perhaps surprisingly, this does not imply that their instruction repetoires facilitate the specification or improve the performance of algebraic algorithms. Analogous to recent work on hardware implementations of the programming language PASCAL [63], investigations to design a non-von Neumann APL machine should be conducted. The resulting product should ultimately be subjected to experimental testing and its performance compared with that of conventional architectures.

Some progress has been made over the past decade in developing new computer architectures to improve the executional performance of algorithms. The most notable successes relating to the area of computational algebra are pipelined architectures (e.g., CRAY-1 [21], Control Data STAR [19], Texas Instruments ASC [58]) and array processors (e.g., ILLIAC IV [4]). Although the state-of-the-art in parallel and distributed logic hardware has advanced rapidly, very little is known about the process of programming in a parallel computing environment. In order to take advantage of the availability of machines with multiple processors, the classical algebraic problems will have to be reexamined and new algorithms devised. But even armed with such algorithms, the process of writing programs which incorporate them will involve the design and development of programming languages (or language extensions) supporting parallelism primitives. In view of the slowness with which structured programming control primitives have been adopted to cope with the analogous problem for single processor systems, this task should not be taken lightly. An ambitious long-range research program exploring the relationship between the performance of parallel algorithms for algebraic computation, the features of programming languages required to express them, and their implementation in hardware would be of obvious benefit.

Symbolic mathematical systems manipulate algebraic formulas directly. These systems are capable of differentiating, integrating, factoring, and simplifying formulas like x^2-1 in addition to performing standard arithmetic operations on such quantities. Examples of symbolic mathematical systems include MACSYMA [45], REDUCE [31], and MATHLAB [26]. All of these systems are quite big and run only on very large machines. Their operation typically involves huge amounts of list processing. Oftentimes, the size of intermediate results obtained before simplification is staggering [46]. Because many of the operations built into these systems involve intricate manipulations of large list structures, the time and storage efficiency of the algorithms used are of paramount importance.

If the convenience and power of such systems are to become available to more users, several aspects of the performance of symbolic mathematical algorithms should be studied further. These include:

1) Studying the features of symbolic mathematical languages. An important issue here is the convenience and generality of system-provided algorithms versus the efficiency of customized routines for particular application areas.
2) Investigating computer architectures to facilitate list processing. A related interesting project is the design and construction of a "symbolic calculator".
3) Examining the computational complexity of symbolic mathematical problems and algorithms, as opposed to purely numerical ones. We note that some of the fast new algorithms for numerical problems (e.g., Strassen's method for matrix multiplication) are less efficient than the classical algorithms for the corresponding symbolic problem. This has been a somewhat neglected area in computational algebra where more basic research is needed.

Several important theoretical questions in the area of algebraic complexity remain open. For example, the "best" matrix multiplication algorithm to date is $O(n^{2.61})$, but with such an extremely high constant of proportionality that the method would be impractical to implement [49]. The strongest lower bounds to date reveal only that order n^2 arithmetic operations are required. Thus, either a better lower bound or an asymptotically far superior matrix multiplication algorithm must exist. Similarly, it is unknown whether the fast Fourier transform (see [9]), whose performance is $O(n \log n)$, is optimal for such applications as polynomial multiplication and the convolution function used in signal processing. The best lower bounds to date are unsurprisingly of order n. Questions such as these have been attacked by leading researchers over the past decade, and no solutions appear to be in sight. Although such fundamental questions are significant, they are not amenable to assault by large research projects. People will continue to work on these key issues anyway, and the questions will more than likely be ultimately resolved by new and unexpected insights.

3.2 Sorting, Searching and Database Systems

Searching and sorting are two of the oldest and best studied problems in computer science. The area of database systems is currently one of the most active. From a theoretical perspective, a database system is a dynamic combination of several different searching algorithms. It is the dynamic (i.e., updatable) nature of the information in a database and the interaction of the various searching algorithms used that make the analysis of database systems difficult.

Knuth [38] is an excellent compilation of searching and sorting algorithms (in isolation, not the interaction between several algorithms). This volume includes experimental comparisons of algorithms to determine crossover points as well as theoretical analyses of numerous algorithms.

One important concern in the analysis of a particular searching or sorting algorithm is whether the retrieval or sort is being done totally within memory or whether I/O is necessary. The appropriate measure of complexity for in-core searches and sorts is the number of comparisons made, whereas the appropriate measure for I/O based searches and sorts is the number of I/O requests made. Another concern is the amount of work required to maintain the data structure for a dynamic set of data. The existence of virtual memory further complicates an analysis since with such systems the distinction between CPU and I/O becomes blurred. The concept of work (i.e., data transferred to and from memory) of Oldehoeft and Bass [48], when broadened to include work done by I/O as well as work done by the CPU, is useful in analyzing this class of algorithms.

The dominant algorithm and data structure that supports retrieval from a dynamic set of data is the B-tree of Bayer and McCreight [6]. A number of variants of this algorithm have been developed to reflect differing application requirements. This algorithm is often used within database systems as one of a series of interacting maintenance and retrieval algorithms. At present, it is unclear how best to structure the interaction between direct retrieval, hashing, and B-trees so as to minimize retrieval I/O requests yet allow maximum flexibility in terms of the logical structure of the database being accessed. This is an area of investigation which should be pursued, both theoretically and experimentally.

One approach to the problem of choosing the optimum combination of access techniques is that proposed by March [44]. He provides a set of equations to predict retrieval times based on retrieval patterns and combinations of access methods used. These equations, if valid, would provide a database designer with the tools needed to appropriately structure the access mechanisms used in a particular database application. An experimental validation effort for March's equations is presently underway [11]. If this effort is to be ultimately fruitful, more analytic work needs to be done to broaden the validity of the assumptions made.

From a language perspective, sorting, searching, and database systems have given rise to the use of nonprocedural human-oriented structures either embedded within existing languages (e.g., the SORT verb in COBOL) or as separate query languages for database systems. These structures are very powerful since they allow great data movement with few commands. This yields a high language level both in the static sense of Halstead [30] and the dynamic sense of Oldehoeft and Bass [48]. It is unclear, however, the extent to which powerful nonprocedural query languages actually simplify the overall solution of problems. Although it may be easier for users to write queries, the search strategies resulting from the use of particular operators may not be clearly understood by the problem solver. A deeper understanding of the underlying data structures and search algorithms is needed if system performance is not to be adversely affected by the use of these languages.

The related problem of extending the Halstead notion of language level to convey the complexity of language operators is a problem worthy of investigation. As an example, consider the relational query languages described in Date [22]. The same problem can be solved with the same data structure using either a query language based on relational algebra or one based on the relational calculus. The former is procedural in nature and similar in spirit to modern algebra, while the latter is nonprocedural and requires the use of quantifiers as in predicate logic. Although both languages have about the same Halstead measures, their psychological complexities (i.e., human comprehensability and usability) are not necessarily the same. Once the factors that underlie psychological complexity are better understood, it should be possible to design better application-oriented languages. Another possibility, which should lead to similar results, is to study various report writing languages with respect to both their features and their dependence on the storage structures of the underlying databases.

The hardware level of sorting and searching again finds expression in the performance of database systems. Parallel schemes for sorting, merging, and selection are discussed in Knuth [38]. Specialized architectures for database systems have recently been proposed and implemented (see, for example, [32]).

The virtues and drawbacks of such architectures need to be examined. Again, the notion of run-time work of Oldehoeft and Bass [48] provides a framework for the evaluation and comparison of various architectural proposals. In this case, the differences between two architectures when solving the same problem (i.e., identical database and search strategy) can be attributed to whether the work (data transformation) is accomplished at the language level, at the run-time level, or at the microcode level.

3.3 Pattern Matching

The pattern matching problem is concerned with the question of whether a given character string, called the subject, contains a specified substring pattern, and if so, locating where in the subject the pattern begins. This problem arises often in processing text of any kind. Applications include macro generators, text editors, word processors, and key-word-in-context information retrieval.

The classical algorithm for this problem is to hold the pattern's leftmost character under the subject's leftmost character and compare. If the two strings match, we are done; otherwise, we slide the pattern one character to the right and try again. Letting n and m denote the lengths of the subject and pattern strings, respectively, this algorithm has a worst case running time proportional to mn since at each of the n characters in the subject we may have to compare all m characters in the pattern (e.g., for a subject 'AAAAAAAA' and pattern 'AAB').

The performance of the classical algorithm is fairly good in actual practice, and pathological strings causing nearly worst case behavior occur quite rarely. Still, Knuth, Morris, and Pratt [39] have devised an asymtotically faster algorithm. Their method, based on finite automata theory, is to preprocess the pattern into a data structure representing a program to search for that one specific pattern, and then apply the program to the subject string. The preprocessing can be performed in time proportional to the pattern's length m, and the program that is produced looks at each of the n characters in the subject string at most once. Hence, the total running time of the algorithm is proportional to m+n in the worst case.

Boyer and Moore [10] have used the basic idea behind the Knuth-Morris-Pratt (KMP) algorithm to derive a substring searching algorithm with an even better average case performance. The KMP algorithm uses i+m character matching operations to locate a pattern beginning at the i-th position in the subject. Boyer and Moore's technique makes it unnecessary to examine every character in the subject, and has been implemented on a PDP-10 computer in such a way that fewer than i+n machine instructions are executed when looking for occurrences of five letter patterns in typical English language text.

The pattern matching algorithms discussed here would serve as an interesting case study for an experimental investigation of algorithm performance. The classical method, probably the choice of most programmers, has lowest "psychological complexity" but highest algorithmic complexity. It would be interesting to see for just which length strings the other methods are better in actual practice. Perhaps the high cost of preprocessing the pattern strings in the KMP and Boyer-Moore algorithms results in excessive overhead, rendering the algorithms infeasible for situations where a pattern is to be used only once or twice, or for strings of the length typically encountered in practice. The results of such experimental investigations would be particularly significant in view of the frequent occurrence of pattern matching applications in nonnumeric information processing. These results could serve to promote the use of the newer, more unfamiliar methods described. Conversely, they might serve as a warning that constant factors are important in actual practice and that asymptotic or order-of-magnitude results must be carefully scrutinized and understood. Another fruitful area of investigation would be to extend the ideas from automata theory serving as the basis for the KMP algorithm to more sophisticated pattern matching operations, like those found in the SNOBOL4 programming language (e.g., alternation, concatenation).

SNOBOL4 [29] is a programming language containing many features not commonly found in other languages. These features greatly facilitate the description of algorithms in several problem areas, most notably applications requiring the manipulation of character string information. Because the facilities of the language are quite different from those provided in conventional computer architecture, elaborate software routines are required to bridge this gap. This means that large amounts of memory are generally used and program execution is quite slow. Thus, the language provides an ideal vehicle for investigating the relationship between the performance of algorithms, programming languages, and computer architecture.

The original implementation of SNOBOL4 [28] was interpretive, with relatively machine-independent source code. To implement the language on any given machine, one wrote machine code for a series of macros. Although this provided a convenient mechanism for getting the language up quickly on a variety of different computers, the resultant implementations were extremely inefficient because of the mismatch between the macro source language and the wide diversity of host machines.

An alternate approach to implementing the language is represented by SPITBOL [24], a compiler that generates machine language code for a particular computer from program code. The compiler approach greatly enhances execution efficiency with little compromise in the SNOBOL source language [25]. But because of the dynamic nature of the language, an extensive run-time library is needed for the elaborate tracing routines, as well as pattern-matching functions. The ratio of the amount of code supporting the run-time environment to the size of the compiler is approximately four to one, a situation atypical of most compiling systems. Observations have shown that an object program generated by the SPITBOL compiler spends a major portion of the execution time in system subroutines. This extensive use of subroutines at run-time again indicates a high discrepency between the features of the source language and those of machine operation.

The convenience of the SNOBOL language for specifying character manipulation algorithms versus the time and storage inefficiency of current implementations justifies looking at alternative ways of implementing the language. Several approaches to the design of a SNOBOL processor have been suggested. Shapiro [56] proposed starting with a traditional von Neumann architecture and adding new fundamental data types and machine instructions to facilitate string processing and recursion. He actually developed a hierarchy of machines, each incorporating more sophisticated hardware structures (e.g., character registers, associative memory), from which the end-user can select the most cost-effective configuration for his application. Chan [14] and Mukhopadhyay [47] have recently proposed non-von Neumann architectures for efficiently implementing the pattern-matching features of the language. Their designs are particularly attractive in light of recent dramatic advances in hardware/firmware technology and rapidly declining hardware costs.

The SNOBOL4 programming language can serve as a case study for investigating most of the major issues addressed in this research plan. The design and implementation of a SNOBOL machine, either in microcode or directly in hardware, can be an experimental means for studying the impacts of computer architecture and programming languages on the performance of algorithms. The time and storage utilization of such a system could be compared with standard implementations. Additionally, production of such a system involves the development of hardware algorithms for nonnumeric computation, an area of algorithmic complexity which has been given altogether too little attention in the past. Finally, Mukhopadhyay [47] notes that many of the features of a SNOBOL machine are required in the front-end of the special-purpose database architectures which are currently being proposed.

3.4 Combinatorial Optimization Problems

This application area encompasses a wide variety of problems such as finding properties of graphs and networks, optimal scheduling, bin packing, set covering and partitioning. Despite the seeming diversity of these problems, similar algorithm design strategies can be used to solve most of them, and interesting relationships between many of the problems have been shown to exist.

The problems which have been studied in the field of combinatorial algorithms can fruitfully be divided into two categories, depending on their worst case execution times. The first class consists of those problems having algorithms whose running time is polynomial in the size of the input. Important examples include finding the shortest distance between two points in a network, the minimum spanning tree problem, maximizing flows in a network, matching and marriage problems, and testing a graph for planarity. To a novice, many of the problems which fall into this category at first seem computationally intractable, requiring essentially exhaustive enumeration procedures for their solution. Usually significant insights into a problem, exploiting some underlying structure, are required before polynomial time algorithms can be devised.

The second class of combinatorial problems are those for which no polynomial time algorithm is known. Algorithms for such problems generally resort to exhaustive enumeration of essentially all possible solutions, and in the worst case have exponential running times. A simple example is the subset sum problem. Here, we are given n positive integers $x_1,\ldots,x_n$ and another positive integer y. We are asked to identify the subset of x_i's whose sum is closest to, but does not exceed, the value of y. There are 2^n subsets of the x_i's, and it appears as though virtually all of these will have to be tested in the worst case. Other important combinatorial problems for which no polynomial time solution currently exists include 0-1 integer programming, the traveling salesman problem, testing for graph isomorphism (equivalence), graph coloring, satisfiability of formulas in propositional logic, and a variety of covering, packing and partitioning problems on sets and graphs.

One might argue that the notion of polynomial time is too imprecise to be used as a criterion for classifying the computational difficulty of a problem. In fact, it has proven to be a very convenient measure. Actually, very few polynomial algorithms with running times of degree greater than 4 or [illegible]ied, although an $O(n^{100})$ procedure would unfortunately

Most of the well-known combinatorial problems which appear to be intrinsically exponential belong to a class called the NP-complete problems, first explored by Cook [20] and Karp [37]. The problems in this class are all computationally equivalent in the sense that if a polynomial time algorithm is found for any one of the problems, then all of them can be solved in polynomial time. Results of this nature are obtained by constructing a polynomial time transformation mapping instances of one problem into equivalent instances of another. Conversely, if an exponential lower bound can be proven for any one of the problems in a sufficiently general model of computation, then all of the NP-complete problems will require exponential time. Most researchers in algorithmic complexity feel that this issue is the most important open question in the entire field. Since this difficult question has been worked on by a large number of prominent researchers, it appears that a satisfactory resolution may not be forthcoming for quite some time.

In view of the fact that instances of NP-complete problems frequently arise in actual computing practice, ways of coping with the apparent intractability of such problems must be devised. One approach is to reduce the search effort as much as possible through the use of branch-and-bound and dynamic programming procedures. The idea is to recognize partial solutions that cannot possibly be extended to actual solutions as soon as possible and eliminate them from further consideration. Although accepting the apparent inevitability of an exponential time solution, use of such procedures can result in substantial time savings. Another approach is to devise algorithms which work quickly for the vast majority of inputs, but resort to more exhaustive means when required.

One fruitful method for dealing with many NP-complete problems is to develop fast approximation algorithms, or heuristics, for their solution. Johnson [35] is a pioneering work in this area. Instead of looking for the optimal solution to an instance of a problem, these procedures seek to find acceptably good solutions, to within specified tolerances, but which operate quickly. Some NP-complete problems can be dealt with satisfactorily in this fashion, but for other problems (e.g., minimal graph coloring), it can be shown that no heuristics exist. Few guiding principles are currently available, and the methods developed to date are very problem specific. This is an important area where further research would clearly be beneficial.

The Euclidean traveling salesman problem is perhaps the most famous example of an NP-complete problem. In this problem, a traveler has to visit each of a number of designated cities on a map and return home via the minimum distance route. All known algorithms which find the shortest tour have a running time which is exponential in the number of cities. In view of the computational infeasibility of finding this exact solution for even a moderate number of points, much attention has been focused on the quality of approximation algorithms for this problem.

Previous researchers have examined the ratio of the path length produced by various heuristic methods in the worst case to that of the optimal route. Rosenkrantz, Stearns, and Lewis [51] have considered several approximation schemes from this perspective. The best approximation algorithm to date for this problem has been developed by Christofides [16]. It has an $O(n^3)$ running time and is guaranteed to find a path whose length is within a factor of 1 1/2 times the optimum. Guarantees of this kind provide a warning about the possible dangers involved with using some particular method. However, such results may be too conservative (pessimistic) since there is experimental evidence that most reasonable approximation schemes perform quite well on the average although there may exist rare maps which force them to find poor tours [36].

Unfortunately, results describing the average behavior of algorithms are generally far more complicated and difficult to derive than those concerning worst case performance. Carney, Kamat, and Lamagna [13] have recently developed techniques using order statistics for examining the expected (average) lengths of paths produced by various approximation methods for the Euclidean traveling salesman problem. Their basic method holds promise for being applicable to other NP-complete problems as well. This and other lines of research aimed toward improved techniques for analyzing the average case behavior of algorithms should be encouraged.

Combinatorial algorithms is another area where experimental investigations into the performance of algorithms would be useful. Although order-of-magnitude analyses have been performed on many combinatorial algorithms, some of the methods proposed appear to be more of theoretical interest than practical value. A beneficial exercise would be to implement competing algorithms for various important combinatorial problems to ascertain their behavior and determine their crossover points. A recent paper by Cheung [15], where the performances of eight algorithms for the maximum flow problem were experimentally compared, is a good example of the type of work which is needed.

Finally, a tutorial paper on combinatorial optimization problems and algorithmic design strategies for their solution would serve a wide audience. Most combinatorial problems are ammenable to solution by several different general methods of attack. Thus, it should be possible to illustrate the use of many of the design techniques listed in the Appendix of this paper by applying each of them to a small number of classical combinatorial optimization problems. Most programmers are not familiar with these general techniques and their wide applicability. They tend to view each algorithm as a special case, and often lack the tools needed to systematically attack unfamiliar problems.

3.5 Computational Geometry

Computational geometry is the area of computer science which deals with the representation of geometric shapes and the solution of problems involving geometric objects on computers. It touches on many other aspects of computer science including, for example, algebraic complexity, graph and network theory, sorting and searching, and computational statistics. Much of the work in this area has been directed toward specific practical goals. The types of problems which have been considered include finding the two nearest neighbors among a set of points, locating the convex hull (or loosely, the perimeter) of a point set, and determining inclusions within and intersections of geometric objects.

Computational geometry is still in its formative stages as a discipline and many of its significant results are new. Because of this, the work which has been done to date has not yet been collected into a concise and cohesive body of knowledge. This is an area where a tutorial or survey paper would be of great benefit. The fact that an n log n algorithm for the convex hull problem was published by Bass and Schubert in 1967 [5], five years before the currently recognized "first" (Graham's algorithm in 1972 [27]), is characteristic of the field.

Shamos completed a doctoral thesis on computational geometry within the framework of the analysis of algorithms in 1978 [55]. To the best of our knowledge, this work is the first systematic approach to the problem. A recent paper by Bentley [7] explores an algorithmic paradigm called multidimensional divide-and-conquer. This technique, which can be used to solve a wide variety of problems, extends previous results to higher dimensional spaces than the plane. These works could easily provide the framework for the needed tutorial, to which motivational applications, detailed algorithms, and recent developments should be added.

It is sometimes convenient to identify two classes of problems in computational geometry: (1) those in which the points or objects are fixed in space, and (2) those in which the points move. Most of the formal work to date has been on problems where the points are fixed. Although some work has been done in the area of changing projections or views of a set of fixed points, little is known about the class of problems involving independently moving points. This latter class includes such practical problems as detecting and tracking independently moving objects.

Aside from the theoretical problems which should be dealt with, there are practical problems of algorithm implementation which require experimental investigations to settle. Here again, we feel there is a need to implement alternative algorithms for common geometric problems and to measure and compare their performances over a wide range of inputs.

Another area for fruitful investigation is the relationship between computational geometry and the various graphics programming languages which are available or have been proposed. It should prove useful to investigate whether one language is more natural than another for stating and solving computational geometry problems.

REFERENCES

[1] A. K. Agrawala and T. G. Rauscher, Foundations of Microprogramming: Architecture, Software, and Applications. Academic Press, 1976.

[2] L. A. Anderson and E. A. Lamagna, "A semantic approach to algorithm analysis", University of Rhode Island, Department of Computer Science and Experimental Statistics, Extended Abstract (April 1980).

[3] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", Communications of the ACM, Vol. 21, No. 8 (August 1978), pp. 613-641.

[4] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnik, and R. A. Stokes, "The ILLIAC IV computer", IEEE Transactions on Computers, Vol. C-17, No. 8 (August 1968), pp. 746-757.

[5] L. J. Bass and S. R. Schubert, "On finding a disk of minimum radius covering a given set of points", Mathematics of Computation, Vol. 21, No. 100 (October 1967), pp. 712-714.

[6] R. Bayer and C. McCreight, "Organization and maintenance of large ordered indexes", Acta Informatica, Vol. 1, No. 3 (August 1972), pp. 173-189.

[7] J. L. Bentley, "Multidimensional divide-and-conquer", Communications of the ACM, Vol. 23, No. 4 (April 1980), pp. 214-229.

[8] A. Borodin and S. Cook, "A time-space tradeoff for sorting on a general sequential model of computation", Proc. 12th Annual ACM Symposium on Theory of Computing (1980), pp. 294-301.

[9] A. Borodin and I. Munro, The Computational Complexity of Algebraic and Numeric Problems. American Elsevier, 1975.

[10] R. S. Boyer and J. S. Moore, "A fast string searching algorithm", Communications of the ACM, Vol. 20, No. 10 (October 1977), pp. 762-772.

[11] R. E. Bunker, "The design and evaluation of a test vehicle for the frame memory model of an information retrieval system", University of Rhode Island, Sc.M. Thesis (August 1980).

[12] C. R. Carlson, "A survey of high-level language computer architecture", in Y. Chu (ed.) High-Level Language Computer Architecture, pp. 31-62. Academic Press, 1975.

[13] E. J. Carney, P. V. Kamat, and E. A. Lamagna, "Expected behavior of approximation algorithms for the Euclidean traveling salesman problem", University of Rhode Island, Department of Computer Science and Experimental Statistics, Technical Report 79-139 (May 1979).

[14] G. K. C. Chan, "A pattern matching processing unit", University of Rhode Island, Sc.M. Thesis (May 1979).

[15] T. Cheung, "Computational comparison of eight methods for the maximum flow problem", ACM Transactions on Mathematical Software, Vol. 6, No. 1 (March 1980), pp. 1-16.

[16] N. Christofides, "Worst-case analysis of a new heuristic for the traveling salesman problem", Carnegie Mellon University, Graduate School of Industrial Administration, Technical Report (1976).

[17] J. Cohen and M. Roth, "On the implementation of Strassen's fast matrix multiplication algorithm", Acta Informatica, Vol. 6, No. 4 (August 1976), pp. 341-355.

[18] J. Cohen and C. Zuckerman, "Two languages for estimating program efficiency", Communications of the ACM, Vol. 17, No. 6 (June 1974), pp. 301-308.

[19] Control Data Corp., "STAR-100 Computer Systems Reference Manual", Publication No. 60256000, Arden Hills, Minnesota (1971).

[20] S. A Cook, "The complexity of theorem proving procedures", Proc. 3rd Annual ACM Symposium on Theory of Computing (1971), pp. 151-158.

[21] Cray Research, Inc., "The Cray 1 Computer Preliminary Reference Manual", Chippewa Falls, Wisconsin (1975).

[22] C. J. Date, An Introduction to Database Systems, Second Edition. Addison-Wesley, 1977.

[23] R. A. DeMillo, R. J. Lipton, and A. J. Perlis, "Social processes and proofs of theorems and programs", Communications of the ACM, Vol. 22, No. 5 (May 1979), pp. 271-280.

[24] R. B. K. Dewar, "SPITBOL Version 2.0", Illinois Institute of Technology (1971).

[25] R. B. K. Dewar and A. P. McCann, "MACRO SPITBOL - a SNOBOL4 compiler", Software - Practice and Experience, Vol. 7 (1977), pp. 95-113.

[26] C. Engelman, "The legacy of MATHLAB 68", Proc. 2nd ACM Symposium on Symbolic and Algebraic Manipulation (1971), pp. 29-41.

[27] R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set", Information Processing Letters, Vol. 1, No. 3 (February 1972), pp. 132-133.

[28] R. E. Griswold, The Macro Implementation of SNOBOL4. W. H. Freeman, 1972.

[29] R. E. Griswold, J. F. Poage, and I. P. Polonsky, The SNOBOL4 Programming Language, Second Edition. Prentice-Hall, 1971.

[30] M. H. Halstead, Elements of Software Science. Elsevier-North Holland, 1977.

[31] A. C. Hearn, "REDUCE 2: a system and language for algebraic manipulation", Proc. 2nd ACM Symposium on Symbolic and Algebraic Manipulation (1971), pp. 128-135.

[32] IEEE Transactions on Computers, Special issue on database machines, Vol. C-28, No. 6 (June 1979).

[33] K. E. Iverson, A Programming Language. Wiley, 1962.

[34] J. Ja'Ja', "Time-space tradeoffs for some algebraic problems", Proc. 12th Annual ACM Symposium on Theory of Computing (1980), pp. 339-350.

[35] D. S. Johnson, "Approximation algorithms for combinatorial problems", Journal of Computer and System Sciences, Vol. 9, No. 3 (December 1974), pp, 256-278.

[36] P. V. Kamat, "Expected behavior of approximation algorithms for the Euclidean traveling salesman problem", University of Rhode Island, Sc. M. Thesis (August 1978).

[37] R. M. Karp, "Reducibility among combinatorial problems", in R. E. Miller and J. W. Thatcher (eds.), Complexity of Computer Computations, pp. 85-104. Plenum Press, 1972.

[38] D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching. Addison-Wesley, 1973.

[39] D. E. Knuth, J. H. Morris, and V.R. Pratt, "Fast pattern matching in strings", SIAM Journal on Computing, Vol. 6, No. 2 (June 1977), pp. 323-350.

[40] D. J. Kuck, "A survey of parallel machine organization and programming", ACM Computing Surveys, Vol. 9, No. 1 (March 1977), pp. 29-59.

[41] D. J. Kuck, The Structure of Computers and Computations, Vol. 1. Wiley, 1978.

[42] E. A. Lamagna, "The complexity of monotone networks for certain bilinear forms, routing problems, sorting, and merging", IEEE Transactions on Computers, Vol. C-28, No. 10 (October 1979), pp. 773-782.

[43] E. A. Lamagna, "Fast computer algebra", University of Rhode Island, Department of Computer Science and Experimental Statistics, Technical Report 80-145 (June 1980).

[44] S. T. March, "Models of storage structures and the design of database records based upon user characterization", University of Minnesota, Ph.D. Thesis (May 1978).

[45] W. A. Martin and R. J. Fateman, "The MACSYMA system", Proc. 2nd ACM Symposium on Symbolic and Algebraic Manipulation (1971), pp. 59-75.

[46] J. Moses, "Algebraic simplification: a guide for the perplexed", Communications of the ACM, Vol. 14, No. 8 (August 1971), pp. 527-537.

[47] A. Mukhopadhyay, "Hardware algorithms for nonumeric computation", IEEE Transactions on Computers, Vol. C-28, No. 6 (June 1979), pp. 384-394.

[48] R. R. Oldehoeft and L. J. Bass, "Dynamic software science with applications", IEEE Transactions on Software Engineering, Vol. SE-5, No. 5 (September 1979), pp. 497-503.

[49] V. Pan, "Field extension and trilinear aggregating, uniting and canceling for the acceleration of matrix multiplication", Proc. 20th Annual IEEE Symposium on Foundations of Computer Science (1979), pp. 28-38.

[50] L. H. Ramshaw, "Formalizing the analysis of algorithms", Stanford University, Ph.D. Thesis (June 1979).

[51] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "An analysis of several heuristics for the traveling salesman problem", SIAM Journal on Computing, Vol. 6, No. 3 (September 1977), pp. 563-581.

[52] J. E. Savage, The Complexity of Computing. Wiley-Interscience, 1976.

[53] J. E. Savage and S. Swamy, "Space-time tradeoffs on the FFT algorithm", IEEE Transactions on Information Theory, Vol. IT-24, No. 5 (September 1978), pp. 563-568.

[54] J. E. Savage and S. Swamy, "Space-time tradeoffs for linear recursion", Proc. 6th Annual ACM Symposium on Principles of Programming Languages (1979), pp. 135-142.

[55] M. I. Shamos, "Computational geometry", Yale University, Ph.D. Thesis (1978).

[56] M. D. Shapiro, "A SNOBOL machine: functional architectural concepts of a string processor", Purdue University, Ph.D. Thesis (June 1972).

[57] V. Strassen, "Gaussian elimination is not optimal", Numerische Mathematik, Vol. 13, No. 4 (August 1969), pp. 354-356.

[58] Texas Instruments, Inc., "A description of the advanced scientific computer system", TI Equipment Group, Austin, Texas, Publication No. 930034-1 (April 1973).

[59] M. Tompa, "Time-space tradeoffs for computing functons, using connectivity properties of their circuits", Proc. 10th Annual ACM Symposium on Theory of Computing (1978), pp. 196-204.

[60] M. Tompa, "Two familiar transitive closure algorithms which admit no polynomial time, sublinear space implementations", Proc. 12th Annual ACM Symposium on Theory of Computing (1980), pp. 333-338.

[61] B. Wegbreit, "Mechanical program analysis", Communications of the ACM, Vol. 18, No. 9 (September 1975), pp. 528-539.

[62] B. Wegbreit, "Verifying program performance", Journal of the ACM, Vol. 23, No. 4 (October 1976), pp. 691-699.

[63] Western Digital Corp., "WD/90 Pascal Microengine Computer", Document No. AO-200, Newport Beach, California (1980).

[64] S. Winograd, "How fast can computers add?", Scientific American, Vol. 219, No. 4 (October 1968), pp. 93-100.

APPENDIX

Analysis of Algorithms and Computational Complexity

Outline and Bibliography

The goal of computational complexity is to quantitatively study the efficiency of algorithms for various tasks performed on a computer. Complexity theory investigates the inherent difficulty of a particular computational problem by deriving good lower bounds on the amounts of various resources, such as time and storage, required for its solution. This provides a framework within which the performances of alternative algorithms for the problem can be compared and improved methods of solution developed.

The major topics in this emerging discipline are listed in the outline which follows. The outline is divided into four principal sections:

1) general issues which delineate the scope of any particular algorithmic complexity investigation,
2) design strategies which have been used to develop new algorithms in diverse application areas and to categorize the problem-solving approaches embodied in most algorithms,
3) methods for deriving lower bounds, or theoretical minima, on the resource requirements to solve a given computational problem independently of the algorithms used, and
4) application areas which have been studied, together with the major problems which have been investigated within these areas.

For those interested in further information, an annotated select bibliography, listing the most important books and survey papers which have appeared, is also provided. Virtually all of the research published to date is accessible through references in the works cited.

Algorithm Analysis and Computational Complexity

An Outline

<u>General Issues</u>

1. Time and space analysis
2. Models of computation
   - Turing machines
   - computer-like models (RAMs and RASPs)
   - decision trees
   - straight-line programs (chains)
3. Exact vs. asymptotic analysis
   - measuring problem size
   - order-of-magnitude (O-, Ω-, Θ- notation)
4. Upper vs. lower bounds
5. Worst case vs. average case

<u>Algorithm Design Techniques</u>

1. Divide-and-conquer (recursion)
2. Greedy method
3. Dynamic programming
4. Basic search and traversal
5. Backtracking
6. Branch-and-bound
7. Approximation algorithms
8. Data structuring

<u>Lower Bounding Methods</u>

1. Trivial lower bounds
2. Decision trees
   - "information-theoretic" bounds
   - oracles and adversary arguments
3. Problem reduction/transformation
   - NP-completeness
4. Algebraic techniques
5. Miscellaneous tricks

<u>Problem Areas</u>

1. Ordering and information retrieval
   - sorting
   - merging
   - selection
   - searching
2. Algebraic and numerical problems
   - evaluation of powers
   - polynomial evaluation and interpolation
   - polynomial multiplication and division
   - matrix multiplication

3. Graphs and networks
   - minimal spanning tree
   - shortest paths
   - connectedness and survivability (connectivity, transitive closure, articulation points, biconnectivity, strong connectivity)
   - circuits (Eulerian, Hamiltonian, traveling salesman problem)
   - graph coloring
   - network flows
   - planarity
   - isomorphism
   - cliques
   - bipartite matching

4. . Computational geometry
   - convex hull
   - closest point problems
   - intersection problems

5. Miscellaneous problems
   - pattern matching in strings
   - cryptography
   - scheduling
   - operations research

Select Bibliography

Books

D. E. Knuth, The Art of Computer Programming (Vol. 1, Fundamental Algorithms; Vol. 2, Seminumerical Algorithms; Vol. 3, Sorting and Searching). Addison-Wesley, 1968, 1969, 1973.

Presents and discusses a wide spectrum of computational problems and algorithms. It is the authoritative source for algorithm theory, and does a nice job on certain aspects of complexity theory (e.g., the treatment of sorting, merging, and selection in Vol. 3). This classic work provides thoroughly comprehensive and historical coverage of its subject matter.

E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms. Computer Science Press, 1978.

A good one-volume introduction to the field. The book is organized around the major algorithm design techniques -- divide-and-conquer, the greedy method, dynamic programming, basic search and traversal techniques, backtracking, branch-and-bound, and algebraic simplification and transformations. Chapters on lower bound theory, NP-completeness, and approximation algorithms are also included.

A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

A more theoretically oriented one-volume overview of the field. Covers topics from a wide variety of problem areas. The book also formulates and compares several computer models such as random access register and stored program machines, and automata-theoretic models (e.g., Turing machines, finite automata, pushdown machines). Contains an outstanding bibliography.

A. Borodin and I. Munro, The Computational Complexity of Algebraic and Numeric Problems. American Elsevier, 1975.

An excellent monograph providing virtually complete coverage of its subject area. Considers such problems as polynomial evaluation, interpolation, and matrix multiplication.

E. M. Reingold, J. Nievergelt, and N. Deo, Combinatorial Algorithms: Theory and Practice. Prentice-Hall, 1977.

Discusses the complexity of a number of important combinatorial problems and analyzes the best known algorithms for their solution. Topics include exhaustive search techniques, generating combinatorial objects, fast sorting and searching, graph algorithms, and NP-hard and NP-complete problems.

M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completness. Freeman, 1979.

Detailed guide to the theory of NP-completeness. Shows how to recognize NP-complete problems and offers practical suggestions for dealing with them effectively. Provides an overview of alternative directions for further research, and contains an extensive list of NP-complete and NP-hard problems.

J. E. Savage, The Complexity of Computing. Wiley-Interscience, 1976.

Covers all of the significant results on the complexity of switching networks, and surveys several other problems in complexity theory. Importantly, this work also attempts to provide a framework for the quantitative study of time-storage tradeoffs and other performance evaluation criteria on models of real computers.

L. I. Kronsjö, Algorithms: Their Complexity and Efficiency. Wiley, 1979.

A mathematically oriented book. Its most important contribution is a detailed discussion of algorithms for numerical problems from the perspective of their numerical accuracy, as well as efficiency. Problems considered include polynomial evaluation, iterative processes, solving sets of linear equations, and the fast Fourier transform. Several nonnumerical applications, most notably sorting and searching, are also discussed.

S. Even, Algorithmic Combinatorics. Macmillan, 1973.

An early treatment of the basic questions explored in combinatorial mathematics. Algorithmic aspects of enumeration problems including generation of permutations and combinations, trees and their properties, and fundamental properties of graphs and networks are considered.

S. Even, Graph Algorithms. Computer Science Press, 1979.

A rigorous treatment of several applications and problems from graph theory. Trees and their properties, graph connectivity and searching, network flows, graph planarity, and NP-completeness are discussed in this monograph.

S. Baase, Computer Algorithms: Introduction to Design and Analysis. Addison-Wesley, 1978.

An upper-level undergraduate text covering selected topics from sorting, graphs, string matching, algebraic problems, relations, and NP-completeness. Aims to develop systematic principles and techniques for studying algorithms. Level of presentation is mathematically thorough.

S. E. Goodman and S T. Hedetniemi, Introduction to the Design and Analysis of Algorithms. McGraw-Hill, 1977.

An undergraduate text, oriented more toward students of programming and less mathematically rigorous than Baase. Like Horowitz and Sahni, this book is organized around the basic algorithm design methods, but its treatment is not nearly as comprehensive (usually one example per technique).

Survey Papers

B. Weide, "A survey of analysis techniques for discrete algorithms", Computing Surveys, Vol. 9, No. 4 (December 1977), pp. 291-313.

A good overview of the field. Discusses all the major issues including models of computation, measuring problem size and asymptotic complexity, lower bounding techniques, worst and average case behavior of algorithms, and approximation methods for NP-complete problems.

J. L. Bentley, "An introduction to algorithm design", Computer, Vol. 12, No. 2 (February 1979), pp. 66-78.

Another good introduction, written primarily for the novice. Contains more illustrative examples than Weide, but does not discuss issues in as much depth. Problems covered include subset testing (via sorting and searching), pattern matching in strings, the FFT, matrix multiplication, and public-key cryptography.

J. E. Hopcroft, "Complexity of computer computations", Proc. IFIP Congress '74, Vol. 3. (1974), pp. 620-626.

Discusses unifying principles in the design of efficient algorithms through the use of several well-chosen examples. More mathematically oriented than some of the other surveys.

E. M. Reingold, "Establishing lower bounds on algorithms -- a survey", AFIPS Spring Joint Computer Conf. '72, Vol. 40 (1972), pp. 471-481.

A clearly written survey of many of the early results concerned with deriving lower bounds on the complexity of functions. Emphasizes ordering (sorting, searching, merging, and selection) and algebraic problems.

R. E. Tarjan, "Complexity of combinatorial algorithms", SIAM Review, Vol. 20, No. 3 (July 1978), pp. 457-491.

Examines recent research into the complexity of combinatorial problems, focusing on the aims of the work, the mathematical tools used, and the important results. Topics covered include machine models and complexity measures, data structures, algorithm design techniques, and a discussion of ten tractable combinatorial problems.

R. M. Karp, "On the computational complexity of combinatorial problems", Networks, Vol. 5, No. 1 (January 1975), pp. 45-68.

A very readable introduction to the theory of NP-completeness.

Finally, the following articles in Scientific American provide a layman's introduction to most of the key issues in the field:

D. E. Knuth, "Algorithms", Vol. 236, No. 4 (April 1977), pp.63-80.

H. R. Lewis and C. H. Papadimitriou, "The efficiency of algorithms", Vol. 238, No. 1 (January 1978), pp. 96-109.

L. J. Stockmeyer and A. K. Chandra, "Intrinsically difficult problems", Vol. 240, No. 5 (May 1979), pp. 140-159.

N. Pippenger, "Complexity theory", Vol. 238, No. 6 (June 1978), pp. 114-124.

M. E. Hellman, "The mathematics of public-key cryptography", Vol. 241, No. 2 (August 1979), pp, 146-157.

R. L. Graham, "The combinatorial mathematics of scheduling", Vol. 238, No. 3 (March 1978), pp. 124-132.

R. G. Bland, "The allocation of resources by linear programming", Vol. 244, No. 6 (June 1981), pp. 126-144.

ATE

LMED

8